BYTE BOOKS™

Programming Techniques

Program Design

Blaise W. Liffick, Editor

Volume 1

Programming Techniques Volume 1
Program Design
Blaise W. Liffick, Editor
BYTE

OTHER BYTE PUBLICATIONS

All **PAPERBYTE®** **Books** contain programs in machine readable object code in the PAPERBYTE® bar code format:

RA6800ML: An M6800 Relocatable Macro Assembler . . . . Jack E. Hemenway
LINK 68: An M6800 Linking Loader . . . Robert D. Grappel & Jack E. Hemenway
TRACER: A 6800 Debugging Program . . Robert D. Grappel & Jack E. Hemenway
MONDEB: An Advanced M6800 Monitor-Debugger . . . . . Don Peters
SUPERWUMPUS . . . . . Jack Emmerichs
Tiny Assembler 6800, Version 3.1 . . . . . Jack Emmerichs
BASEX: A Simple Language and Compiler for 8080 System . . . . . Paul Warme
K2FDOS: A Floppy Disk Operating System for the 8080 . . . . . Ken Welles
BAR CODE LOADER . . . . . Ken Budnick

Other **BYTE BOOKS,**™ collections of favorite articles from past issues of **BYTE** magazine, plus new material and addenda:

Programming Techniques: Simulation . . . . . Blaise W. Liffick (ed.)
Programming Techniques: Numbers in Theory and Practice . . . . . Blaise W. Liffick (ed.)
Programming Techniques: Bits and Pieces . . . . . Blaise W. Liffick (ed.)
BASIC Scientific Subroutines, Volume I . . . . . Fred Ruckdeschel
Ciarcia's Circuit Cellar . . . . . Steve Ciarcia
The BYTE Book of Computer Music . . . . . Christopher P. Morgan (ed.)
The BYTE Book of Pascal . . . . . Blaise W. Liffick (ed.)
Beginner's Guide for the UCSD Pascal System . . . . . Kenneth L. Bowles

Programming Techniques

Volume I

Program Design

Blaise W. Liffick, Editor

BYTE Publications, Inc.
70 Main Street
Peterborough, New Hampshire 03458

Library of Congress Cataloging in Publication Data
Main entry under title:

Programming Techniques: Program Design.

1.Electronic digital computers — Programming I. Liffick, Blaise W.
QA76.6.P7518 001.6'424 78-8649
ISBN 0-07-037825-8

Printed in the United States of America

TABLE OF CONTENTS

FROM THE EDITOR

This book begins a new effort by BYTE Publications to provide our readers with the best available manuscripts on the major topics of interest to the home computerist. Included in the new series of BYTE's books are reprints of the best articles from past issues of BYTE magazine, plus new material which has not been printed anywhere before. The books will be organized in logical volumes of related topics. This provides the reader with vital information from previous BYTE issues which he or she might have missed, new material that has not appeared in the magazine, plus a book covering one specific theme for quick, easy reference.

Manuscripts included in these books are of the same high quality as those found in BYTE magazine, because we use the same stringent criteria in selecting new manuscripts for inclusion in these books as we do in choosing them for the magazine. Generally, the additional criterion used to select manuscripts for the books instead of the magazine is a constraint on the length of articles used in the magazine itself. In addition, we receive so many quality manuscripts that we could never possibly include them all in BYTE magazine. Therefore, in our efforts to give the reader all the information needed to be a successful microcomputerist, we have decided to make these manuscripts available in book form.

The book that you are holding in your hands is the first in a series on the general topic of Programming Techniques. This particular book deals with the details of the theory behind the design of the various aspects of programs. Anyone who has programmed for any length of time will agree that the most critical part of writing a program of any kind (application, system software, etc) is in the design phase, both the initial specifications and the program logic design. The actual coding of the program amounts to more of a mechanical process once the initial design of the program has taken place. Therefore, it is easy to see that unless the original design of the program is correct, the program cannot be expected to work as per specifications.

The purpose of this book, then, is to provide the personal computer user with the techniques needed to design efficient, effective, maintainable programs. Included in the topics covered are structured program design, modular programming techniques, program logic design, and examples of some of the more common traps the casual, as well as the experienced, programmer may fall into. In addition, details on various aspects of the actual program functions, such as hashed tables and binary tree processing, are included.

Further books in this series will make available new techniques and further developments of the existing ones as they occur. This will allow you, the personal computer user, to stay up to date with the current technology of programming skills.

Blaise W. Liffick
Editor

PROGRAM STRUCTURE

About This Section

For the last several years, those of us whose profession has been programming (applications, systems, scientific, whatever) have been bombarded on all sides with the latest philosophies of programming: structured, modular, top down, bottom up, GOTOless, etc. Not only do we get encouragement from employers to embrace whatever the most current popular technique for coding is, but we also get it from others in our profession who are adherents to one or the other philosophy. This is not to imply that any or all of the techniques do not have merit, but most of the coding philosophers are talking only about just that: coding. The main thrust of their basic arguments is against poor coding practices. And that's just fine. But they forgot one important detail: once a program has been designed, all the coding techniques in the world are generally ineffective because the major portion of the program logic has already been set! The specifications and initial design of the program predetermines to a great extent how the coding can be performed.

In the following section, the techniques for *designing* effective programs are presented. Both the amateur and professional programmer will profit from these practical techniques of design by being able to produce essentially error-free code. And for the amateur programmer there is an added bonus for following these practices: instant documentation! By carefully designing the function of the program before ever coding a single line, you insure that once the coding *is* completed, you can add something at any time. The code written so long ago will be easily understood, and you will know where and how to make any necessary changes.

In addition, if everyone followed similar guidelines for designing programs, trading programs would be a painless and easy way to expand your program library. You could instantly understand what anyone else's program was doing. And while someone's 6800 code definitely will not run on your 8080 machine, the program has already been totally designed and can be easily coded into any other language!

Structured Program Design

David A Higgins

In the world of electronics, no experimenter in his right mind would build a circuit by throwing a few parts together with some wire and some hope, then attaching a line cord and plugging it in to see if it works. Not only are you likely to destroy some very expensive parts, but it is also a good way to get fried, or at least get a new hairdo.

Yet, after all the trouble that a serious microcomputer hobbyist will take to insure that his circuit is put together correctly before he ever turns it on, he will invariably try to program his new computer by using a technique analogous to the one above. That is why his programs almost never run right the first time, if indeed they ever manage to run right at all. It is also why many microcomputer buffs stay up until odd hours of the night drinking coffee by the gallon in an effort to find that one little bug.

But there is hope. I'm sure that nearly everyone involved with computers has heard something about structured programming in one form or another. It is not really a new technique, having been preached about for many years. However, the tools and methodologies available to design programs have changed radically over the years.

In the beginning there were flowcharts, which looked like five-dimensional octopi or the corporate structure of a conglomerate. Despite the absence of a consistent approach that would enable everyone to design a program using flowcharts, those programmers who did bother to work out

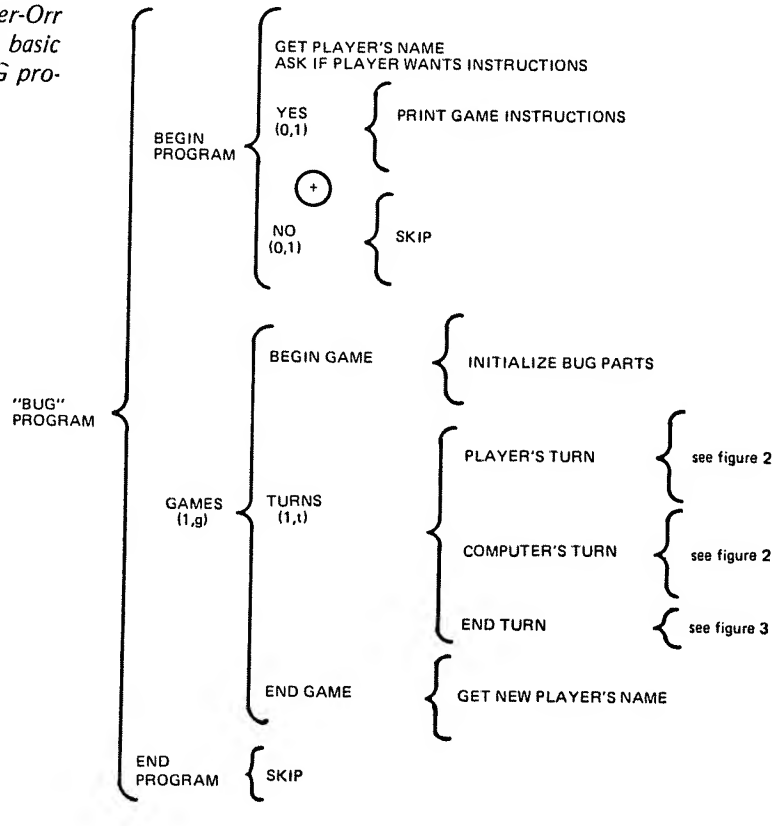


Figure 1: The Warnier-Orr diagram showing the basic structure of the BUG program.

their problem with a flowchart first usually seemed to have more luck in getting programs to run sooner and better than programmers who did not.

Structuring Tools

The development of mathematics would surely have been stymied if Roman numerals had been retained as our number system. In much the same manner, the science of structured program design would have been mired down if only flowcharts had been available for developing programs. It is not that calculus is impossible with Roman numerals, it's just that it's extremely difficult. Thus, over the years, a number of design and documentation tools were developed to better enable a programmer to understand the problem before going out to do battle with the program.

TOP-DOWN or GOTO-less programming, developed by Dijkstra and others, was probably the first major attempt to solve the design versus coding problem. Dijkstra simply observed that the more GOTOs that were in a program, the less likely it was to run correctly. Dijkstra called such programs "spaghetti bowl" programs, because if you drew a line from each GOTO in the program to its destination, you ended up with a mess that looked like a bowl of spaghetti. He showed how any program could be written with just a few simple flow structures without any GOTOs. His techniques produced simple, readable code that was easy to test and maintain. So, the big push among design aficionados was to eliminate the GOTOs in their programming. Although TOP-DOWN programming was a big advancement over flowcharting, it was just that: programming. It was a technique for coding a program, not necessarily designing it.

Another technique, IBM's HIPO (and later HIPO-DB) entered the design field almost by chance, being primarily a documentation tool that was also being used for program design. The major drawback to HIPO techniques, besides the fact that they did not work well for designing a program, was their tendency to produce 50 pages of documentation for a three page program.

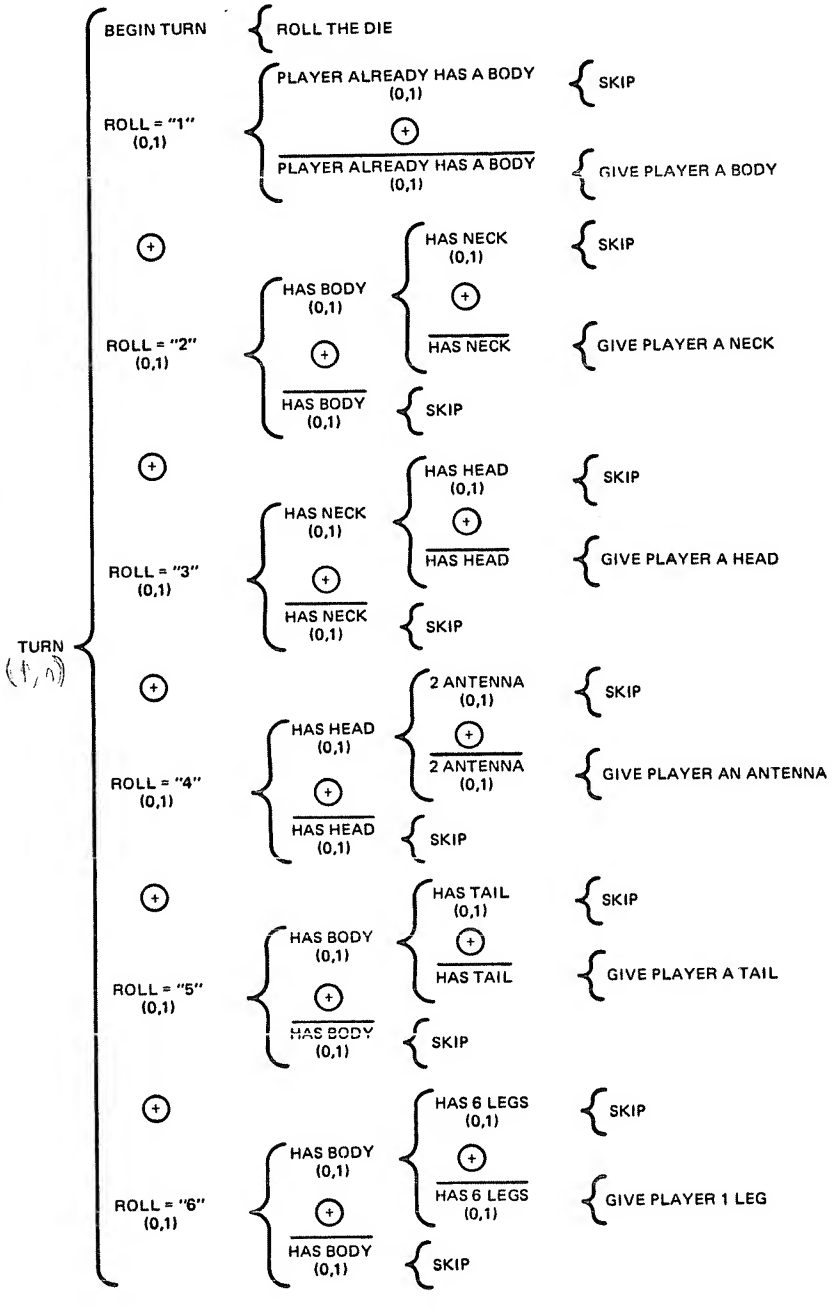


Figure 2: Diagram of the logic for the PLAYER and COMPUTER TURNS routines of the BUG program. Note that item means "the complement of item."

Warnier-Orr Diagrams – A New Approach

Within the last four years a new technique for program design has evolved from the work of Jean-Dominque Warnier (pronounced warn'-yay) in France, and Kenneth T Orr of Langston, Kitch and Associates in Topeka KS. The technique has foundations in set theory and Boolean algebra, and holds much promise for program design applications. Warnier-Orr diagrams, as we have called them here in the United States, allow programmers to design faster than ever before, to code programs with little or no effort, and produce programs that usually run correctly the first time. The approach is not limited to small programs. Nothing will make a believer out of someone quicker than a 20 page COBOL program which runs correctly the first time. The Warnier-Orr technique stresses design over coding and

contends that once a problem is designed, it does not matter what programming language you code it in! At Langston, Kitch and Associates, people have used the technique to program in COBOL, PL/I, ALGOL, FORTRAN, BASIC, RPGII and assembler languages. It works equally well for all of them.

Warnier-Orr Diagram

The simplest way to learn about Warnier-Orr diagrams is to see examples of them. Warnier-Orr diagrams are very easy to learn and use; however, be forewarned that this is a technique that is sometimes deceptively simple, but not as trivial as it often seems.

Let's consider the relatively simple game of BUG. In this game the computer rolls a die, once for itself and once for its opponent. Each number of the die corresponds to a part of the BUG's anatomy: 1 = BODY, 2 = NECK, 3 = HEAD, 4 = ANTENNAE, 5 = TAIL, and 6 = LEGS. The object of the game is to finish your bug before the computer finishes its bug. Other rules: you must have a body before you can have legs, a neck or a tail; you must have a neck before you can have a head, and you must have a head before you can have antennae. One body, one neck, one head, one tail, six legs and two antennae are needed to complete a bug. Figure 1 is a Warnier-Orr diagram showing the basic structure of the BUG program.

The Warnier-Orr diagram is read left to right, top to bottom, just like conventional English text. The brackets enclose logically related operations, the largest of which is the program itself. The BUG program is composed of three logical sections:

- The BEGIN PROGRAM section, where the player's name is requested and there is an explanation of the game rules. Note that the ⊕ symbol between the modules YES and NO denotes the exclusive OR function, meaning that one or the other but not both of the modules will be performed. Observe also that this is reflected in the number of times that each module may be performed: 0 if the condition is false and 1 if the condition is true.
- The process section, GAMES, where the playing of the game actually takes place. The (1,g) denotes that the section is to be performed at least once, and possibly many (g) times.
- The END PROGRAM section, which in this case is empty, but which usually contains things such as the closing of files, the goodbye message, etc.

The rest of the brackets decompose in a similar fashion. The GAMES procedure breaks down into the beginning of the game, (BEGIN GAME), the turns that each player takes (TURNS), and the end of the game (END GAME).

Notice that logically there are things that only happen at the beginning of the program and things that only happen during the playing of the game itself. The Warnier-Orr di-

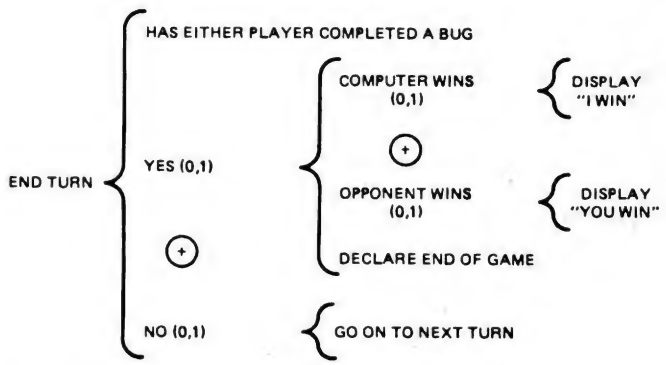


Figure 3: Warnier-Orr diagram for the ending of a turn or a game.

Listing 1: A structured BASIC program that was written using the Warnier-Orr diagrams of figures 1 thru 3. This code executed correctly the first time even though it was the author's first attempt at writing a BASIC program.

```
10    REM BUG PROGRAM
20    REM BEGIN PROGRAM
30    DIM HEAD(2), BODY(2), LEGS(2), TAIL(2), ANTE(2), NECK(2), CNT(2)
40    GOSUB 120
50    REM GAMES (1,G)
60    LET EPGM=0
70    GOSUB 200
80    IF EPGM=0 THEN GOTO 70
90    REM END PROGRAM
100   STOP
110   REM BEGIN PROGRAM SUBROUTINE
120   PRINT 'ENTER YOUR FIRST NAME'
130   INPUT :NAME$
140   PRINT 'DO YOU WANT AN EXPLANATION OF THE RULES; ENTER YES
      OR NO.'
150   INPUT ANS$
155   LET TEST = SCOMP ('YES' ,ANS$)
160   IF TEST = 0 THEN GOSUB 1200 ELSE ;
170   RETURN
180   REM GAMES SUBROUTINE
190   REM BEGIN GAME
200   GOSUB 290
210   REM TURNS (1,T)
220   LET EGAM = 0
230   GOSUB 390
240   IF EGAM = 0 THEN 230
250   REM END GAME
260   GOSUB 1150
270   RETURN
280   REM BEGIN GAME SUBROUTINE
290   LET BODY(1), BODY(2) = 0
295   LET CNT(1), CNT(2) = 0
300   LET NECK(1). NECK(2) = 0
```

Listing 1, continued:

```
310   LET HEAD(1), HEAD(2) = 0
320   LET ANTE(1), ANTE(2) = 0
330   LET TAIL(1), TAIL(2) = 0
340   LET LEGS(1), LEGS(2) = 0
350   RETURN
360   REM TURNS SUBROUTINE
370   REM PLAYERS TURN
380   REM LET PLAYER START TURN
390   PRINT 'HIT RETURN TO ROLL DIE'
400   INPUT A
410   LET PLAY = 1
420   GOSUB 520
430   REM COMPUTERS TURN
440   LET PLAY = 2
450   GOSUB 520
460   REM END TURN
470   GOSUB 1060
480   RETURN
490   REM TURN SUBROUTINE
500   REM   PLAY=1;PLAYERS TURN—PLAY=2;COMPUTERS TURN
510   REM ROLL DIE
520   LET ROLL = FIX@ (((RND (0))  * 6.0)) + 1
530   PRINT : "ROLL IS A ", ROLL
540   IF ROLL = 1 THEN IF BODY (PLAY) #1 THEN GOSUB 690 ELSE ; ELSE ;
550   IF ROLL=1 THEN 650
560   IF ROLL = 2 THEN IF BODY (PLAY) = 1 THEN IF NECK (PLAY) # 1
      THEN GOSUB 760
570   IF ROLL=2 THEN 650
580   IF ROLL = 3 THEN IF BODY (PLAY) = 1 THEN IF NECK (PLAY) = 1
      THEN IF HEAD (PLAY) #1 THEN GOSUB 820
590   IF ROLL=3 THEN 650
600   IF ROLL = 4 THEN IF HEAD (PLAY) = 1 THEN IF ANTE (PLAY) # 2
      THEN GOSUB 880
610   IF ROLL=4 THEN 650
620   IF ROLL = 5 THEN IF BODY (PLAY) = 1 THEN IF TAIL (PLAY) # 1
      THEN GOSUB 940
630   IF ROLL=5 THEN 650
640   IF ROLL = 6 THEN IF BODY (PLAY) = 1 THEN IF LEGS(PLAY)
      # 6 THEN GOSUB 1000
650   RETURN
670   REM BODY SUBROUTINE
700   IF PLAY = 1 THEN PRINT : NAME$, " 'S BUG HAS A HEAD"
710   IF PLAY = 2 THEN PRINT : "COMPUTER'S BUG HAS A HEAD"
720   LET CNT (PLAY) = 1
730   LET BODY (PLAY) = 1
740   RETURN
750   REM NECK SUBROUTINE
760   IF PLAY = 1 THEN PRINT : NAME$, " 'S BUG HAS A NECK"
770   IF PLAY = 2 THEN PRINT : "COMPUTER'S BUG HAS A NECK"
780   LET CNT (PLAY) = CNT (PLAY) + 1
790   LET NECK (PLAY) = 1
800   RETURN
810   REM HEAD SUBROUTINE
820   IF PLAY = 1 THEN PRINT : NAME$, " 'S BUG HAS A BODY"
830   IF PLAY = 2 THEN PRINT : "COMPUTER'S BUG HAS A BODY"
840   LET CNT (PLAY) = CNT (PLAY) + 1
850   LET HEAD (PLAY) = 1
860   RETURN
870   REM ANTENNAE SUBROUTINE
880   LET ANTE(PLAY) = ANTE(PLAY) + 1
890   IF PLAY = 1 THEN PRINT : NAME$, " 'S BUG HAS ",
      ANTE (1), " ANTENNAE."
900   IF PLAY = 2 THEN PRINT : "COMPUTER'S BUG HAS", ANTE (2)
      " ANTENNAE."
910   LET CNT (PLAY) = CNT (PLAY) + 1
920   RETURN
930   REM TAIL SUBROUTINE
940   IF PLAY = 1 THEN PRINT : NAME$, " 'S BUG HAS A TAIL"
950   IF PLAY = 2 THEN PRINT : "COMPUTER'S BUG HAS A TAIL"
960   LET CNT (PLAY) = CNT (PLAY) + 1
970   LET TAIL (PLAY) = 1
980   RETURN
990   REM LEGS SUBROUTINE
1000  LET LEGS(PLAY) = LEGS(PLAY) + 1
1010  IF PLAY = 1 THEN PRINT : NAME$, " ' S BUG HAS ", LEGS (1), " LEGS."
1020  IF PLAY = 2 THEN PRINT : "COMPUTER'S BUG HAS ", LEGS (2),
      "LEGS."
1030  LET CNT (PLAY) = CNT (PLAY) + 1
1040  RETURN
1050  REM END TURN SUBROUTINE
1060  IF CNT (1) = 12 THEN 1090
1070  IF CNT (2) = 12 THEN 1110
1080  GOTO 1130
```

agrams allow you to see very easily just where and when a particular event must take place. After examining figure 1 carefully to make sure that you understand how the diagrams work, move on to the explanation of the PLAYER and COMPUTER TURNS section shown in figure 2.

In figure 2, we have represented the logic for each of the players' turns during the game. At the beginning of each turn, the die is rolled to determine the part of the BUG's body that the player may receive. Whatever the roll, we then have a logical path to follow. Again, please note that the presence of the ⊕ between each of the possible rolls denotes mutual exclusion, ie: only one of the paths may be selected. This particular structure is known as a case statement.

If the player rolls a 4, we first find the instructions to follow for a roll of 4 and check to see if the player has a BUG head. If he does, we then check to see whether or not the player already has two antennae. If he does, then we do nothing. If he does not have two antennae yet, we give him one antenna. If he does not have a BUG head, then again we do nothing. In a similar fashion, all of the possible rolls and their associated procedures are explained. Now let's move on to the Warnier-Orr diagram for the end of the turn, which is shown in figure 3.

If either player has won the game at the end of a turn, the computer declares the winner and ends the game. If neither player has won, the computer does nothing and cycles through for another turn.

Structured Programming

Having fully understood the problem, coding the BUG program is a simple and straightforward process. For this particular example I coded the program shown in listing 1 in a version of BASIC.

As you can see, each bracket of the original Warnier-Orr diagram roughly corresponds to a subroutine in the finished code: the process GAMES, for instance, becomes the subroutine at line number 180 which is called repeatedly by the branch at line 80 until EPGM equals 1, indicating that no more games are to be played; the process BEGIN PROGRAM is handled by the subroutine at line 110, and so forth. The resultant code is:

- easy to read and understand
- easy to change and maintain
- already documented
- logically correct.

It is also a program that will run correctly the first time, barring unforeseen syntax

errors for those of us who can't type or spell. All of this is possible because the program was thoroughly designed before it was even partially coded.

Conclusion

Warnier-Orr diagrams are a giant leap in the right direction for structured programming. They represent an attitude which, for the first time since people have been playing with computers, can lead to consistently reliable software that is very easy to maintain. Currently, most data processing departments spend over 80% of their time and effort repairing old code that has suddenly gone bad. Warnier-Orr diagrams also provide the means to produce software of a quality that has never before been possible.

If you think that you are interested in using Warnier-Orr diagrams to help you solve some of your software headaches, by all means try them. But as I mentioned above, this technique looks deceptively simple, and you may not have much success. Understanding a diagram such as the one presented in this text is one thing; creating one from scratch is another.

If you do get bogged down, please feel free to write us for more information. If you try them, like them, and think you've done something exciting with them, again feel free to write us and tell us what you've done.■

Listing 1, continued:

```
1090 PRINT : NAME$, " 'S BUG IS FINISHED' YOU WIN"
1100 GOTO 1120
1110 PRINT : "COMPUTER'S BUG IS FINISHED, I WIN"
1120 LET EGAM = 1
1130 RETURN
1140 REM END GAME  SUBROUTINE
1150 PRINT : "DOES ANYONE ELSE WANT TO PLAY"
1160 INPUT ANS$
1165 LET TEST = SCOMP (ANS$, 'YES')
1170 IF TEST # 0 THEN LET EPGM = 1
1180 RETURN
1190 REM EXPLANATION OF RULES SUBROUTINE
1200 PRINT "THE GAME OF BUG IS PLAYED AS FOLLOWS:"
1210 PRINT " A DIE IS ROLLED BY THE COMPUTER, AND EACH NUMBER"
1220 PRINT " ON THE DIE CORRESPONDS TO A PART OF THE BUG'S "
1230 PRINT " BODY: 1=BODY, 2=NECK, 3=HEAD, 4=ANTENNAE, 5=TAIL"
1240 PRINT " 6=LEGS. YOU NEED 1 BODY, 1 NECK, 1 HEAD, 2 ANTENNAE"
1250 PRINT " 1 TAIL, AND 6 LEGS TO COMPLETE A BUG."
1260 PRINT " THE OBJECT OF THE GAME IS TO BUILD YOUR BUG
     BEFORE"
1270 PRINT " COMPUTER BUILDS HIS."
1280 PRINT " —HIT RETURN WHEN YOU ARE READY TO PLAY."
1290 INPUT A
1210 RETURN
```

BIBLIOGRAPHY

1. Orr, Kenneth T, *Structured Systems Development*, Yourdon Press, New York, 1977.

2. Warnier, J D, *Logical Construction of Programs*, Van Nostrand Reinhold Company, New York, 1976.

Structured Programming with Warnier-Orr Diagrams

David A Higgins

Part 1: Design Methodology

Any successful program design methodology must be able to do several things: it must produce consistent, low cost, high reliability results; it must produce them quickly, while still allowing for easy maintenance later and, it must be simple enough to allow anyone (and I do mean anyone) to use it. Warnier-Orr diagrams (after Jean-Dominique Warnier in France and Kenneth T Orr in the United States) satisfy all of the above requirements with an added bonus; they produce structured programs that nearly always run correctly at the first effective trial. They allow people to produce superprograms without being superprogrammers.

The purpose of this article is to show how to develop and code a structured program using the Warnier-Orr methodology from start to finish. The technique is a straightforward approach to producing correct programs. It is just as valid and successful for personal microcomputer applications as it is for megacomputer applications in the world of business, science and industry. I feel that this method of designing a program is one of the most advanced state of the art software development techniques in existence today. It is a concise, step by step method with predictable results.

Step One: Identify the Output

This is the first, the primary and the most important rule of all for the construction of a correct program. It cannot be emphasized enough. The failure to **first** identify the outputs of a program is usually the primary reason programs fail to run correctly.

You must ask yourself the questions: "How will I be able to tell when I am through with this program?" "What will the printed, displayed and punched outputs physically look like?" "What will the program be able to do?" All of these questions must be thoroughly answered before you can even begin to think of coding the program. Skipping this step because "Aw, I know what I want to do," or "Gee, this isn't any fun, let's start coding," is a common mistake, and although you may get away with it on a small program once in a while, omitting it will kill you more often than not.

A good example of the kind of trouble you can get into by assuming that you know everything about a problem can be found in a recent popular film. In the movie *Jeremiah Johnson*, Jeremiah befriends an old hunter and trapper in the mountains. The old hunter asks Jeremiah if he can skin a bear. "Of course I can," he replies. In the next scene, we see the old man running down a hill towards the cabin closely pursued by a very large bear. The hunter runs into the open front door, leaps out of the back window and yells: "There . . . you skin that one and I'll go get you another." Jeremiah failed to do one basic thing; he forgot to ask whether the bear he was supposed to skin was dead. Skinning a dead bear is one thing, skinning one that is still running around the room trying to skin you is quite another. Just as writing a program after it has been properly defined is one thing, and trying to write one when you aren't even sure what it is supposed to do when you are finished is another.

Defining outputs is not really an unreasonable requirement to make; after all, no building contractor would begin construction without first knowing what the finished building was supposed to look like; no electrical engineer would start soldering

parts together without a schematic diagram. In fact, no profession (reliable profession anyway) involved in the business of putting things together ever starts to build anything unless they know what it will look like after they are done. Yet, that is precisely the way most programmers try to write programs. Then they wonder what went wrong when they have problems. The same programming principles which apply to the professional apply just as much to the amateur, for no one's time is unlimited.

After defining all of the outputs of the program, the next step is to define the logical data base, although you will probably never really spend much time at this step with most personal microcomputer applications.

MONTHLY FINANCIAL REPORT
FOR THE MONTH OF JANUARY 1977

DATE	CHECK#	TO:	DEBIT	CREDIT	BALANCE
		BALANCE FORWARD OF			$231.90
1	978	GROCERY STORE -MILK, BREAD, EGGS	2.23		229.67
1	979	PHONE COMPANY	37.14		192.53
3	980	GAS BILL	25.61		166.92
5	981	GEORGE FREDRICK -SHOVELLING SNOW	5.00		156.92
5		PAYCHECK DEPOSIT		312.18	469.10
6	982	ELECTRIC COMPANY	23.15		445.95
		. . .	. . .	. . .	. . .
31	1013	BYTE MAGAZINE -SUBSCRIPTION RENEWAL	12.00		237.11
		CURRENT BALANCE			237.11

Figure 1: Proposed output of a computer program for balancing a checkbook and producing an end of month report.

Step Two: Define the Logical Data Base

The reason this step is trivial for many personal use applications is because the logical data base typically consists of only one numeric field. It is typically the field holding a person's response to a program generated question. For illustrative purposes let us look at a home computer application that requires a slightly more complex data base arrangement. Take for instance a computer program that would balance the family checkbook and produce a financial report each month. The report designed in step one might look something like figure 1.

If you were keeping manual records that you wanted to be able to search very easily, you would keep each one of those entries, perhaps on index cards, filed by year, by month and by date. Figure 2 illustrates a way of representing the logical data structure for the checkbook balance report in Warnier-Orr notation.

In figure 2, you can see the logical data structure for the checkbook balance report. The report is organized by year; within each year by months; within each month by days; and within each day by transactions, which are either debits (checks) or credits (deposits). Note that year, month, day, and transactions all appear in the report at least once and possibly many times; thus we see the notation (1,n) in the diagram. Having an entry for a day that had no transactions or having a monthly report with no days is hardly worth the trouble. However, each transaction is either a credit transaction (credit occurring once, and debit not occurring) or a debit transaction (debit occurring once and credit not occurring). This condition is reflected on the chart by the "⊕" symbol, which is the symbol for mutual exclusion.

One important point needs to be made here. The diagram of figure 2 is *not* the logical data base for this report; it is only the report's logical data structure. Making a chart of the logical data base requires that we map the data elements that appear in the report onto the logical report structure, as we have done in figure 3. In figure 2 we showed conceptual relationships of one part of the structure to another. In figure 3 we've filled in the required details needed to complete each level of the structure. One level of the structure corresponds to one bracket and the levels are counted left to right.

Step Three: Define the Physical Data Base

Defining the physical data base of a program is largely a packaging decision: what physical arrangement of the data in the computer will best suit the needs of the program. The only help I can give you on this is the simple suggestion that the physical

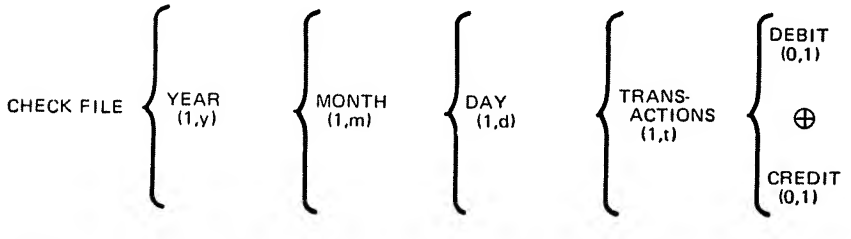


Figure 2. Logical data structure for the checkbook balance report. The notation (1,n) indicates an operation will take place at least once and up to n times, inclusive.

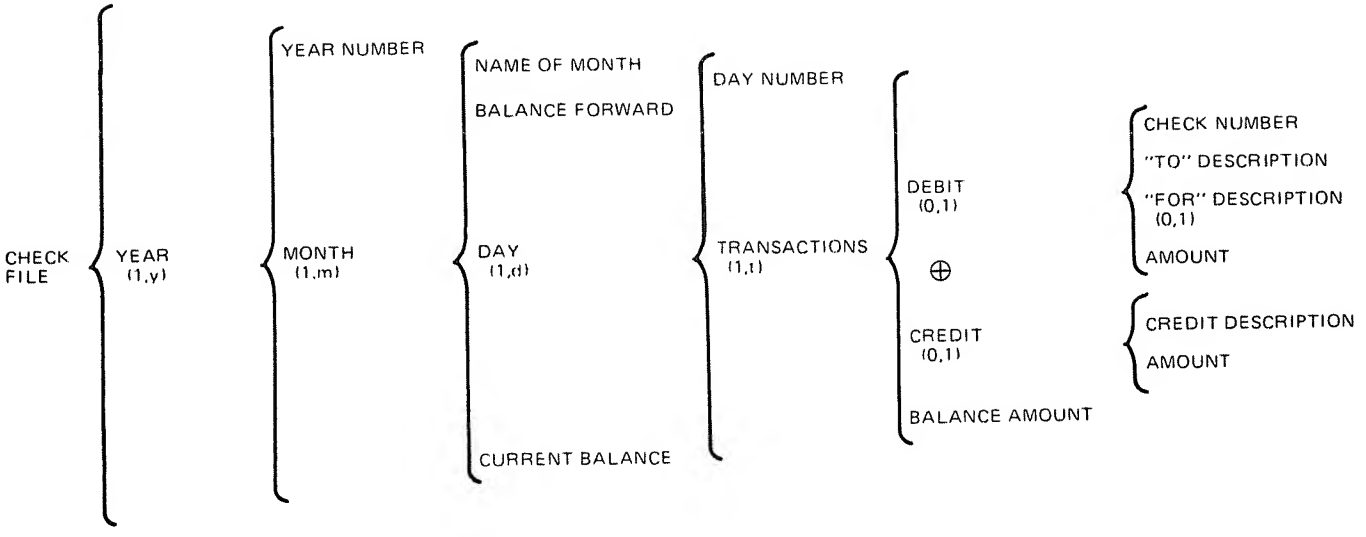


Figure 3: The logical data base is generated by mapping the data elements that appear in the report onto the logical data structure.

representation should mirror the logical representation in all but the most extreme cases. These are hardware decisions. You may wish to construct a file one way if you are using a cassette tape storage system; you may construct it another way if you have a floppy disk. You would not want to impose a file structure that forced a cassette tape to behave like a disk by running back and forth through the tape at high speed. That is a good way to burn up a tape drive in a hurry. Ultimately, as memories become faster, more versatile and more efficient, the physical data base will probably always be able to mirror the logical data base. Magnetic bubble memories, for instance, have no moving parts to burn up.

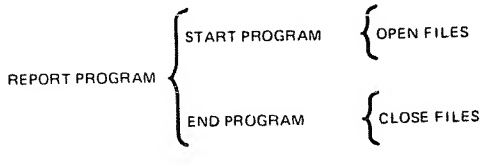
In the checkbook balance report program the simplest physical data base would be a sequential file. The necessary information and a brief description of each transaction could be stored in the order shown in figure 4, read left to right.

Given that we have a file with this information on it which is sorted by year, month, day and transaction, producing a report program is almost a trivial exercise.

Step Four: Design the Process Structure

Since in this case we are working with a single program, the process structure will ultimately represent the program structure. Were we designing an entire system, an accounts receivable system for instance, the process structure would represent many programs and the associated system procedures that would operate them. The process structure is obtained from the same logical data structure that the logical data base was derived from.

Referring again to both figures 1 and 2, we can begin to design the program from the bottom to the top. Looking first at the leftmost bracket of figure 2, which for this step is labeled REPORT PROGRAM, we could draw a structure thus:



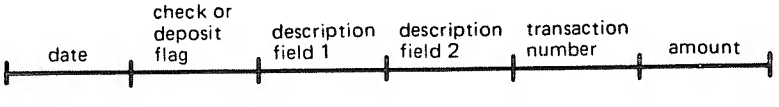
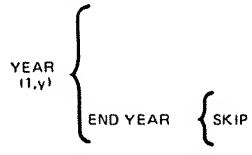


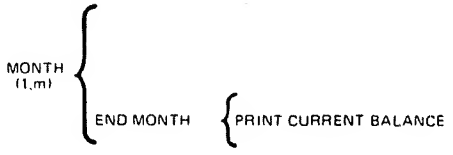
Figure 4: A sequential file with a record format such as this is the simplest physical data base for the checkbook program. The information that is needed has been decided by the logical data base. The order they are put on the file depends on exactly what you intend to do. Since in this case we will be sorting by date, the date of the transaction appears first on the file.

Note that program structure is denoted by left to right positioning, and that sequences of operations are noted top (first) to bottom (last).

We can see that the only thing for us to do at the beginning of the program is to open the files, and the only thing to do at the end of the program is to close the files we have used. Moving right to the YEAR bracket, the process END YEAR must be defined. For this program there is nothing to do at the end of the year, so we fill in the bracket with the notation SKIP:



For the bracket labeled MONTH, there is the matter of printing the CURRENT BALANCE at the end of the month:



There are no processes to be performed at the end of each DAY, therefore we show the END DAY process the same way as the END YEAR process:

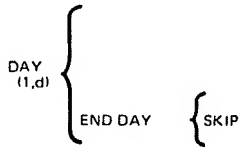
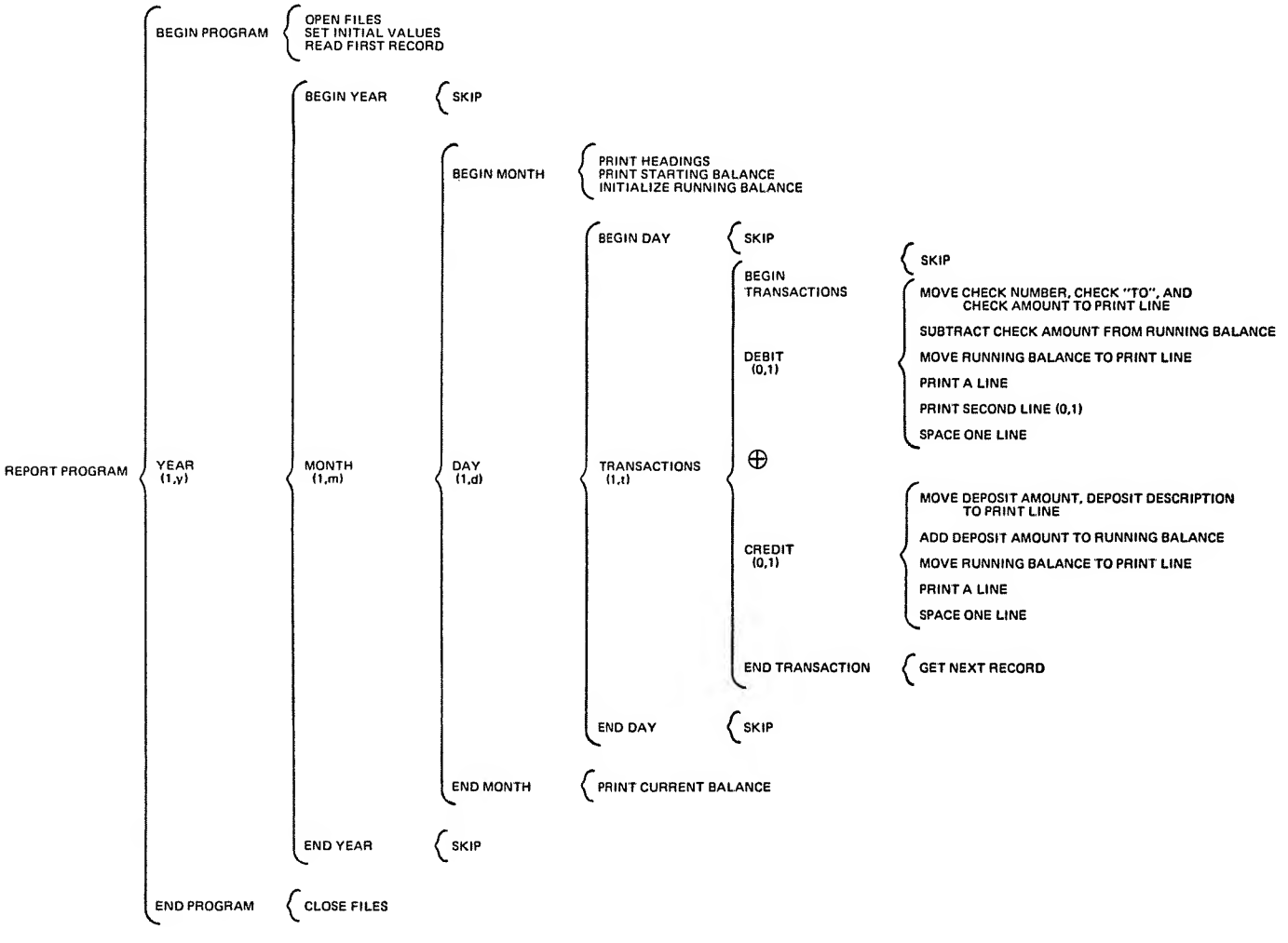
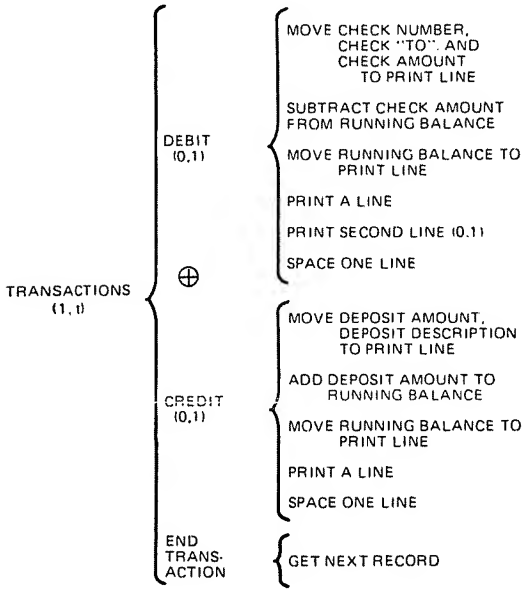


Figure 5: Completed Warnier-Orr diagram for a checkbook balancing report program. This program arrangement will probably result in the smallest amount of memory being used. The sequences of operations at any given level (left-right position) are read from top to bottom. A level of operations corresponds to a logical level of procedure calls in a block structured programming language.



The TRANSACTIONS process is where most of the work is done. For each CREDIT or DEBIT, one line and possibly a second (for DEBIT) is printed, showing the appropriate information; the running balance is updated, and the next record must be read:



With this much of the program design done, the only things to be filled in are the BEGIN brackets for each level. The entire diagram with these processes added is shown in figure 5.

Looking at the Warnier-Orr diagram for the checkbook balance program, you can see the entire series of events which must take place to correctly process the report as it was given. Note also that this is the only correct structure that will produce the checkbook balance report. Any other structure that will produce the report is isomorphic to this structure. The structure is also optimal in operation, in the sense that nothing is ever done unless it must be done.

The program which is coded from this structure will also have some predictable features. It will run as quickly as possible. It will usually require the least amount of storage. It is very easy to maintain, and it will run correctly at the first effective trial. Not bad dividends for a half hour of extra work. Syntax runs are not effective trials, but, with a little diligence and effort, syntax errors can also be brought under control.

Part 2 will show how easy it is to fill in the details of structured programs using Warnier-Orr diagrams.■

Structured Programming with Warnier-Orr Diagrams

David A Higgins

Part 2: Coding the Program

In part 1 we carefully constructed a design structure. In order to make the most of that structure a few words about programming style are in order. While it is true to a certain extent that any method of coding the structure will produce a logically correct program, matters of syntactical errors resulting from shoddy coding techniques as well as problems with maintenance seem to indicate that a great deal of care should be exercised in the construction of the actual program code.

For this particular example, I'll use a fairly standard version of BASIC that

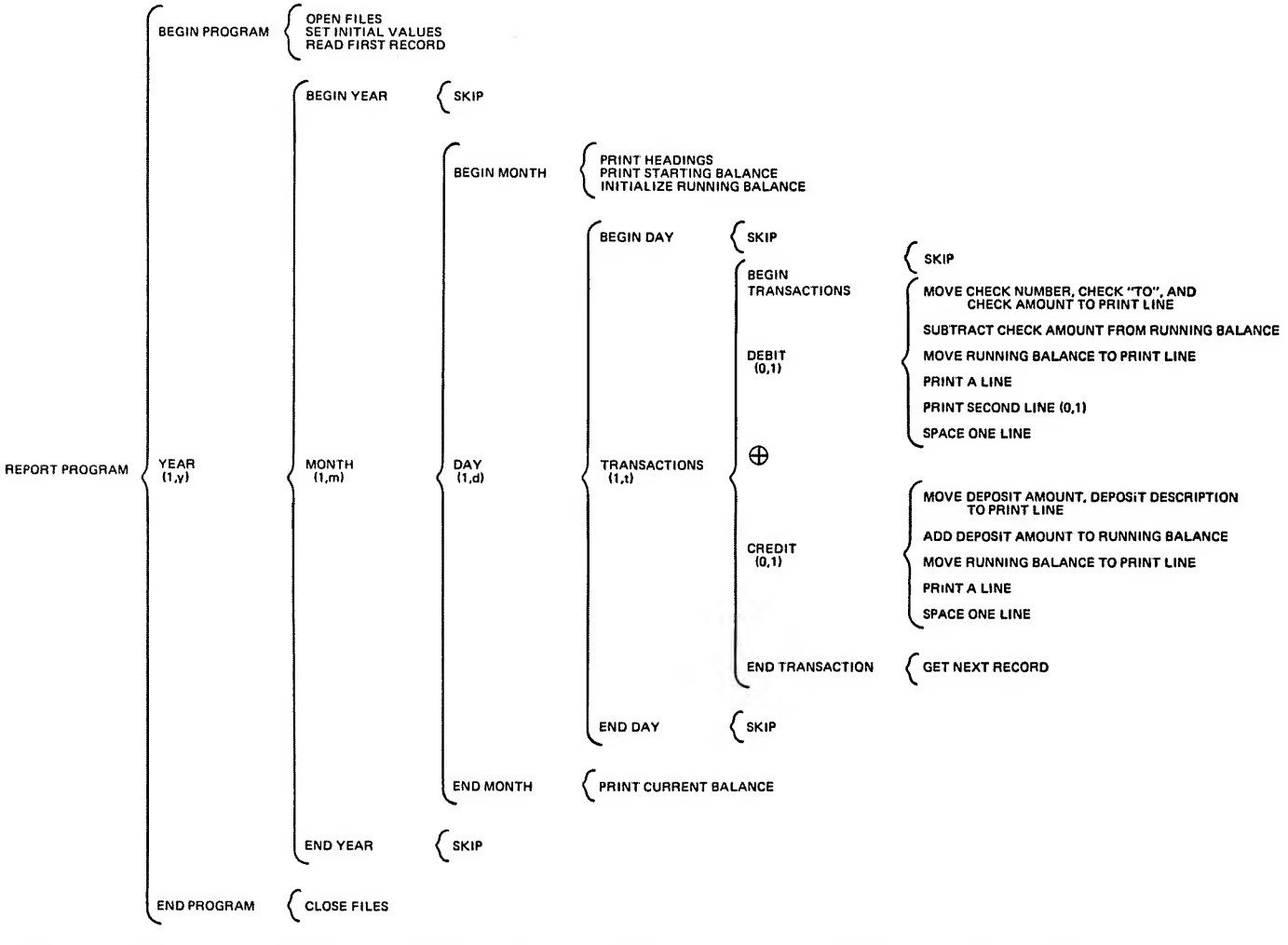


Figure 1: Final Warnier-Orr diagram description of the checkbook balance report program (reproduced from part 1).

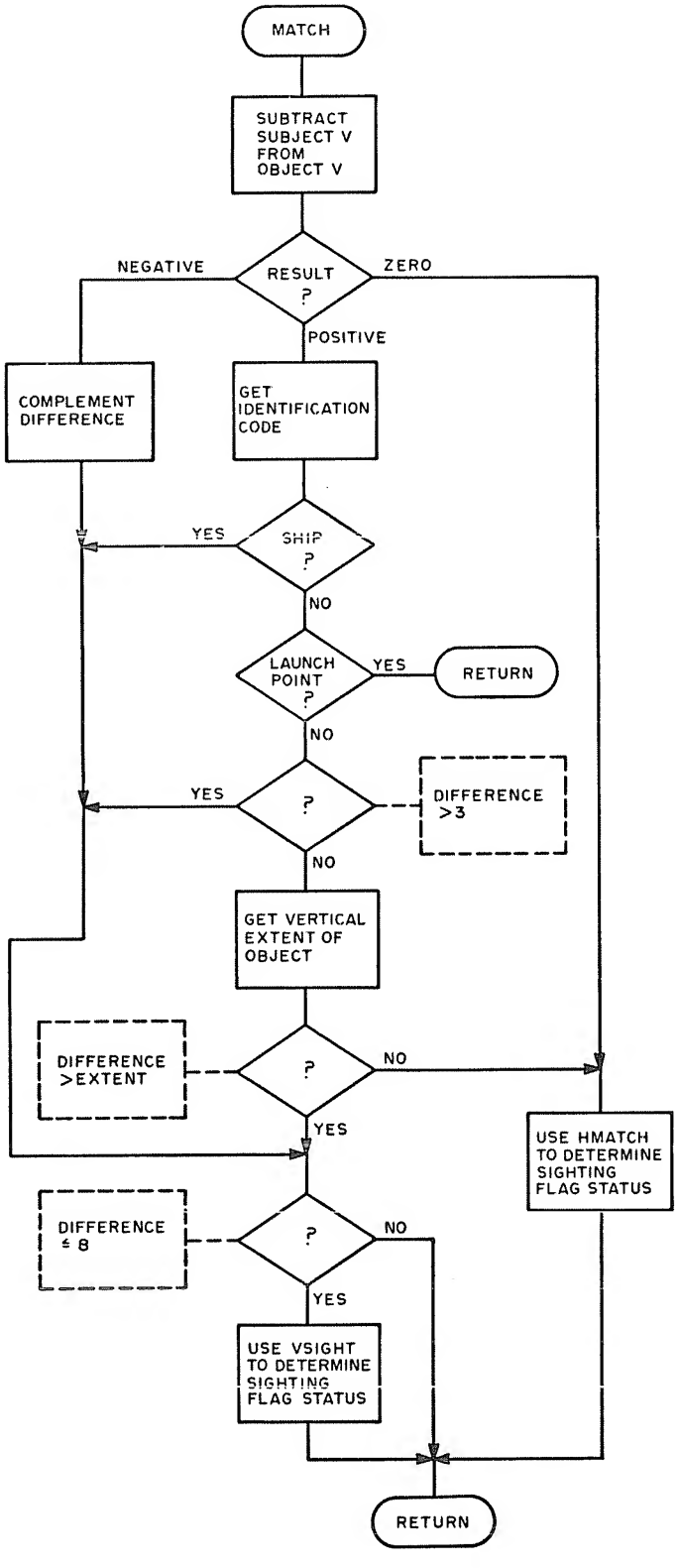


Figure 2: This a a flowchart chosen at random for comparison to a Warnier-Orr representation.

runs on a J100 Jacquard Systems computer. The concepts and construction rules are just as applicable to Tiny BASIC, assembly language, and especially APL. Obeying the following five coding conventions will help you write a program that will execute on the first time.

Coding Convention 1: Names Should Be Indicative of Function

For versions of BASIC that only allow one letter names, this is often a little hard, but for most other languages with multiple character symbols, it is a must. For instance, a field that contains an amount should be labeled AMOUNT, an address field should probably be called ADRESS, and so forth. Cutesy names: SNEEZY, DOPEY, GRUMPY, HELL (a perennial favorite label for adolescent COBOL programmers) are to be strictly avoided.

Coding Convention 2: Comments Should Be Used Freely

Comment lines in programs written in obscure languages, APL for instance, should probably outnumber actual lines of code. Comment lines are especially useful for explaining unclear methods of calculation, complex decisions, etc.

Coding Convention 3: Every Bracket of a Warnier-Orr Diagram Should Represent a New Subroutine

Languages that do not permit subroutines or languages that limit the levels of nesting of subroutines are very tricky to use and should be avoided if at all possible. Save your spare change for three or four weeks and go buy a better version of BASIC; there are plenty of good ones on the market. In BASIC, each subroutine should be clearly labeled with REMark statements.

Coding Convention 4: Subroutines Should Be as Short as Possible

If a subroutine contains too many statements it is difficult to understand and maintain. It also means you are probably doing something in this subroutine that should be put in another subsequent subroutine. In most high level languages a practical limit of 10 to 20 statements is appropriate. This rule is standard structured programming practice.

Coding Convention 5: GO TOs Should Be Avoided

In higher level languages, GO TOs can often and should be eliminated entirely. However, in versions of BASIC that do not

have a DO verb and in assembler, GO TOs are often necessary. Utmost care is urged whenever a GO TO is used; it should only be used as a last resort. In assembly language, use of arbitrary jumps or branches should be avoided.

When coding the program, the order of the subroutines is not crucial. The only piece of code that must be fixed in any certain location is the highest level bracket which must be the first executable line, or lines, of code. One possible way of coding the first section is to omit the first bracket and consider the code as the main program. For BASIC, subroutine calls are left unnumbered until the subroutine is actually written. In this case, we use nnn to indicate an unknown number.

```
100 REM CHECKBOOK BALANCE REPORT PROGRAM

110 REM     BEGIN PROGRAM
120         GOSUB nnn

130 REM     YEAR (1,Y)
140         LET ENDYR = FALSE
150         GOSUB nnn
160         IF ENDYR = FALSE THEN GOTO 150

170 REM     END PROGRAM
180         GOSUB nnn
190         END
```

Another way to program this section would be to have the above piece of code as a subroutine to an even higher level procedure as follows.

```
80 REM CHECKBOOK BALANCE REPORT PROGRAM
90 GOSUB 110
95 END
   .
   . 100 through 180 as above
   .
200 RETURN
```

Either way of coding is acceptable. Note that the GO TO in statement 160 is used to create the structure of a DO UNTIL, a feature that is not available with this particular BASIC.

The center path of the Warnier-Orr diagram is the easiest to begin to code at this point. So the code for the YEAR, the MONTH, and the DAY routines is shown next; for the subroutine YEAR:

```
250 REM YEARLY PROCEDURE

260 REM     BEGIN YEAR
270         GOSUB nnn

280 REM     MONTHS (1,M)
290         LET ENDMO = FALSE
300         GOSUB nnn
310         IF ENDMO = FALSE THEN GOTO 300

320 REM     END YEAR
330         GOSUB nnn
340         RETURN
```

For the subroutine MONTH:

```
350 REM MONTHLY PROCEDURE

360 REM     BEGIN MONTH
370         GOSUB nnn

380 REM     DAYS (1,D)
390         LET ENDAY = FALSE
400         GOSUB nnn
410         IF ENDAY = FALSE THEN GOTO 400

420 REM     END MONTH
430         GOSUB nnn
440         RETURN
```

For the subroutine DAY:

```
450 REM     DAILY PROCEDURE

460 REM     BEGIN DAY
470         GOSUB nnn

480 REM     TRANSACTIONS (1,T)
490         LET ENDTRN = FALSE
500         GOSUB nnn
510         IF ENDTRN = FALSE THEN GOTO 500

520 REM     END DAY
530         GOSUB nnn
540         RETURN
```

The TRANSACTIONS process breaks down as follows:

```
550 REM     TRANSACTIONS ROUTINE

560 REM     CREDIT (0,1) OR DEBIT (0,1)
570 IF CDFLAG = CREDIT THEN GOSUB nnn ELSE GOSUB nnn

580 REM     END TRANSACTION
590         GOSUB nnn
600         RETURN
```

Subroutine DEBIT is coded a bit differently from the way it was designed for one simple reason. BASIC will let you output from the same fields that were read in as input; many languages do not. Therefore, the only code remaining in the subroutine is the subtraction of the amount from the running balance and the print commands.

```
610 REM DEBIT PROCEDURE
620     LET RUNBAL = RUNBAL - AMOUNT
650     PRINT ON PRINTR: DAY, CHKNUM, DESC1, DRAMT, CRAMT, RUNBAL
660     IF DESC2 # SPACES THEN PRINT ON PRINTER: DESC2
670     PRINT ON PRINTR: SPACES
680     RETURN
```

The symbol # is the not equal to operator. Note that this code makes no attempt to format the output line. Although the facility is available with this version of BASIC, it differs greatly from other line formatting BASICs around, and would serve only to confuse the immediate issue.

The CREDIT process is very similar to the DEBIT process.

```
690 REM     CREDIT PROCEDURE
700         LET RUNBAL = RUNBAL + AMOUNT
730         PRINT ON PRINTR: DAY, DESC1, CRAMT, DRAMT, RUNBAL
740         PRINT ON PRINTR: SPACES
750         RETURN
```

The only remaining subroutines to be coded appear below:

```
760 REM     END TRANSACTION
763         LET OLDDAY = DAY
770          INPUT FROM CHECK1: DAY, CDFLAG, DESC1, DESC2,
             CHKNUM, $ AMOUNT
775         ON ENDFILE GOSUB nnn
```

continued on page 24

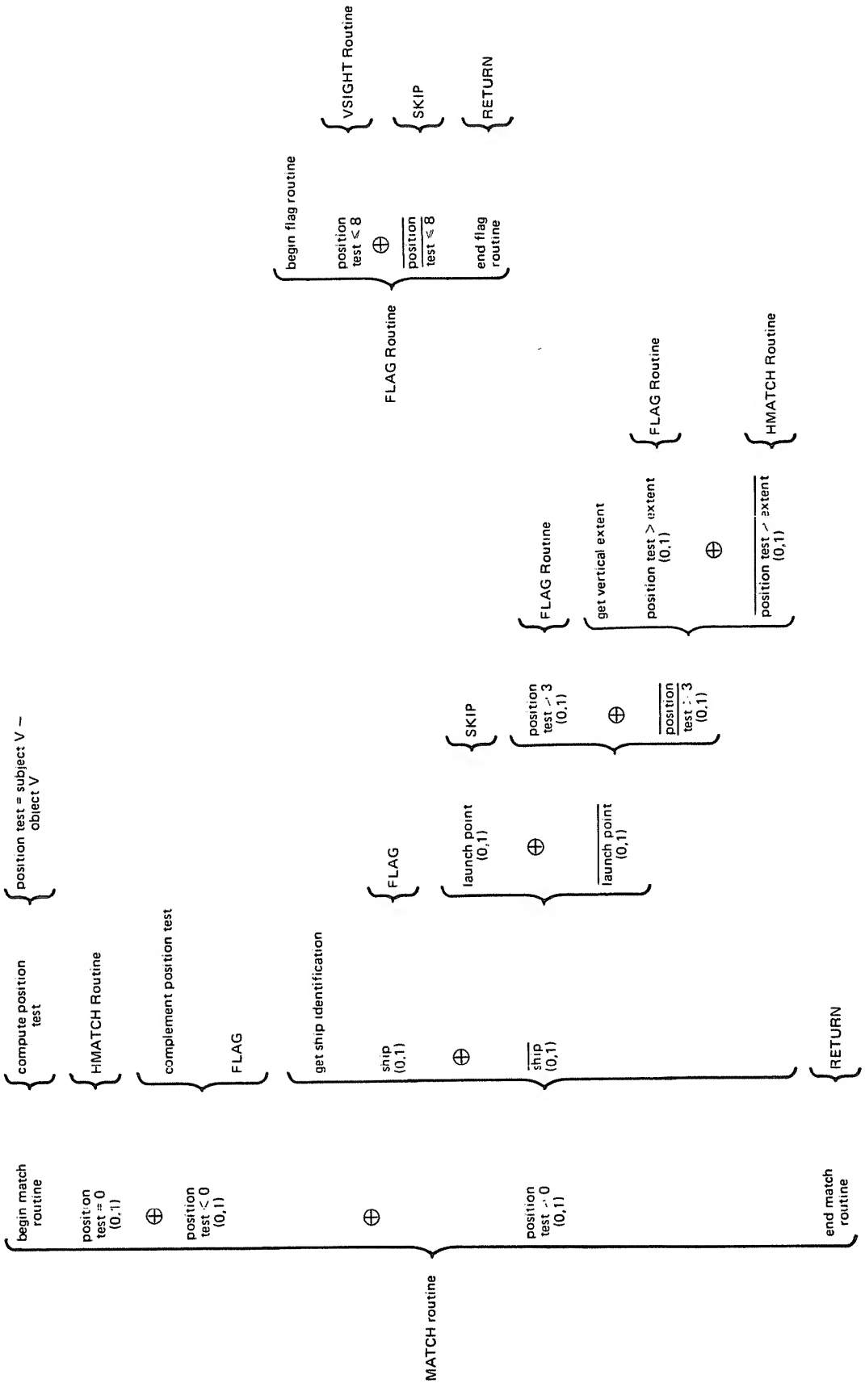


Figure 3: The original flowchart in figure 2 converted into Warnier-Orr diagram. This is a much simpler looking diagram and is easier to follow and explain to someone. Since it is broken down into sections it can be programmed as a series of subroutines that can be easily maintained and modifed. Note that item means "the complement of item."

Listing 1: BASIC source listing for the checkbook balance report program. Each of the subroutines can be matched with one of the brackets in the diagram of figure 1. The individual modules that do not contain any code should be left as they are to facilitate easy maintenance in the future.

```
100 REM CHECKBOOK BALANCE REPORT PROGRAM
110
120 REM     BEGIN PROGRAM
130         GOSUB 1090
140
150 REM     YEAR (1,Y)
160         LET ENDYR = FALSE
170         GOSUB 280
180         IF ENDYR = FALSE THEN GOTO 170
190
200 REM     END PROGRAM
210         GOSUB 1290
220         END
230
240 REM ***************************************
250 REM YEARLY PROCEDURE
260
270 REM     BEGIN YEAR
280         GOSUB 1470
290
300 REM     MONTH (1,M)
310         LET ENDMO = FALSE
320         GOSUB 430
330         IF ENDMO = FALSE THEN GOTO 320
340
350 REM     END YEAR
360         GOSUB 1390
370         RETURN
380
390 REM ***************************************
400 REM MONTHLY PROCEDURE
410
420 REM     BEGIN MONTH
430         GOSUB 1210
440
450 REM     DAYS (1,D)
460         LET ENDAY = FALSE
470         GOSUB 580
480         IF ENDAY = FALSE THEN GOTO 470
490
500 REM     END MONTH
510         GOSUB 1340
520         RETURN
530
540 REM ***************************************
550 REM DAILY PROCEDURE
560
570 REM     BEGIN DAY
580         GOSUB 1500
590
600 REM     TRANSACTIONS (1,T)
610         LET ENDTR = FALSE
620         GOSUB 720
630         IF ENDTR = FALSE THEN GOTO 620
640
650 REM     END DAY
660         GOSUB 1430
665         RETURN
670
680 REM ***************************************
690 REM TRANSACTIONS PROCEDURE
700
710 REM     CREDIT (0, 1) OR DEBIT (0, 1)
720         IF CDFLAG = DEBIT THEN GOSUB 800 ELSE GOSUB 890
730
740 REM     END TRANSACTION
750         GOSUB 965
760         RETURN
770
775 REM ***************************************
780 REM DEBIT PROCEDURE
790
800         LET RUNBAL = RUNBAL - AMOUNT
810         PRINT :DAY;CHKNUM;DESC1;'     ';AMOUNT;RUNBAL
820         IF DESC2 # ' ' THEN PRINT :SPACES;DESC2
830         PRINT :SPACES
840         RETURN
850
860 REM ***************************************
870 REM CREDIT PROCEDURE
880
890         LET RUNBAL = RUNBAL + AMOUNT
900         PRINT :DAY'     ';DESC1;AMOUNT;'     ';RUNBAL
910         PRINT :SPACES
920         RETURN
930
940 REM ***************************************
950 REM END TRANSACTION
960
965         LET OLDDAY = DAY
970         INPUT FROM CHECKS:DAY,CHKNUM,CDFLAG,DESC1,DESC2,AMOUNT
980         ON ENDFILE CHECKS GOSUB 1030
985         IF OLDDAY # DAY THEN LET ENDTR = TRUE
990         RETURN
1000
1010 REM ***************************************
1020 REM END OF FILE
1025
1030        LET ENDAY, ENDMO, ENDTR, ENDYR = TRUE
1040        RETURN
1050
1060 REM ***************************************
1070 REM BEGIN PROGRAM PROCEDURE
1080
1090        OPEN 'CHECKS',SYMBOLIC,INPUT:CHECKS
1100        STRING SPACES, CDFLAG, DESC1, DESC2, MONTH
1110        DECIMAL AMOUNT, BALANC, RUNBAL
1120        LET TRUE = 1
1130        LET FALSE = 1
1140        LET SPACES = '
1150        INPUT FROM CHECKS: DAY, CHKNUM, CDFLAG, DESC1, DESC2,
            AMOUNT, BALANC, & MONTH, YEAR
1160        RETURN
1170
1180 REM ***************************************
1190 REM BEGIN MONTH
1200
1210        PRINT :'          CHECKBALANCE REPORT'
1220        PRINT :'      FOR THE MONTH OF ':MONTH;YEAR
1230        PRINT :SPACES,'BALANCE FORWARD OF ';BALANC
1235        LET RUNBAL = BALANC
1240        PRINT :'DAY CHECK# FOR          DEBIT CREDIT BALANCE'
1250        RETURN
1260
1265 REM ***************************************
1270 REM END PROGRAM
1280
1290        CLOSE CHECKS
1300        RETURN
1310
1315 REM ***************************************
1320 REM END MONTH
1330
1340        PRINT :'CURRENT BALANCE ',RUNBAL
1350        RETURN
1360
1365 REM ***************************************
1370 REM END YEAR
1380
1390        RETURN
1400
1405 REM ***************************************
1410 REM END DAY
1420
1430        RETURN
1440
1445 REM ***************************************
1450 REM BEGIN YEAR
1460
1470        RETURN
1480
1485 REM ***************************************
1490 REM BEGIN DAY
1495
1500        RETURN
```

continued from page 21

```
778         IF OLDDAY # DAY THEN LET ENDAT = TRUE
780         RETURN

790 REM     END OF CHECK FILE DEFAULT SUBROUTINE
800         LET ENDAY, ENDTR, ENDMO, ENDYR = TRUE
810         RETURN

820 REM     BEGIN MONTH PROCEDURE
830         PRINT ON PRINTR: HDR1$
840         LET RUNBAL = BALANC
850         PRINT ON PRINTR: RUNBAL
860         PRINT ON PRINTR: HDR2$
870         PRINT ON PRINTR: SPACES
880         RETURN

890 REM     END MONTH PROCEDURE
900         PRINT ON PRINTR: RUNBAL
920         RETURN
```

The program is finished with the BEGIN PROGRAM and the END PROGRAM subroutines, which are not developed here, and the replacing of the untagged GOSUBs coded before. The modules for which a GOSUB was generated should probably remain a part of the program even though they contain no code. They make maintenance much easier. The entire working program with formatting and other embellishments appears in listing 1.

Conclusion

The art of programming has become a process which can be taught to anyone who needs to use it, which is something that we have not been able to accomplish until very recently. Admittedly, the technique for developing programs presented here is sometimes tedious and not very creative, but it will get the job done. In the personal computer field a lot of enthusiasts probably enjoy programming on the fly and spending all night debugging. But for those who don't, including myself, and who aren't satisfied with just running someone else's canned programs, there is an alternative. As the pioneer in this methodology, Jean-Dominique Warnier, puts it: "If you don't have time to do it right, do you have time to do it over?" Realistically, one cannot say that this methodology is the ultimate in software process design or that it is completely right. It is not. Something is sure to come along in the future that is better. But, for now, it is certainly a large step in the right direction.■

Once I finished reading about the ease with which Warnier-Orr diagrams could be used I decided to take a sample flowchart and convert it into the Warnier-Orr form to see how much of a difference there actually was. I happened to be working on an article by Geoffrey Gass (entitled "Starfleet") which contained a large number of flowcharts. Choosing one at random I converted it. Figure 2 is the original flowchart. Figure 3 is the converted diagram. I think the Warnier-Orr form is much easier to read and understand.

When designing with flowcharts it is sometimes difficult not to cross lines or have a great deal of redundancy in the program which makes it difficult to follow. All the arrows going across the paper are very distracting and hard to follow. The Warnier-Orr diagram does not have this disturbing problem. It is very easy to follow the program through the various subroutines.

The Warnier-Orr diagram lends itself to structured program writing. If you consider each of the separate brackets another subroutine it is very easy to write the program just as it stands from top to bottom. When we use conventional flowchart techniques we end up leaping about the program to perform statements that are at various parts of the same routine. In my opinion the Warnier-Orr diagram is a quantum leap in the direction of aid for structured program designers.

Ray Cote
Editor
BYTE Publications

Warnier-Orr Diagrams: Some Further Thoughts

GT Wedemeyer

The article "Structured Program Design" in the October 1977 BYTE, page 146[1], has certainly simplified my thinking. However, the use of the symbol ⊕ seems to violate a rule implicit in the Warnier-Orr diagram that one need not and in fact must not go up in a list contained within a bracket of a given order. The ⊕ symbol requires checking up and down the list of case statements. I believe that what is meant is illustrated in figure 1. In this example CASE J is equivalent to ROLL = "J." This manner of diagramming clarifies the relationship between statements having alternatives and statements not having alternatives. It also eliminates the need for the instruction SKIP, since the finding of no more items in a list of a given order is the equivalent of an instruction to return to the proper place in the list of the next lower order, where the order of a list is its position from left to right as shown in figure 2.

I would like to define the instruction RETURN to mean "in the list of next lower order than the list in which this instruction is found, complete the step immediately following the lowest completed step." Although this instruction seems implicit, as I indicated above, I would prefer that it be explicitly stated, and I think it would make the diagrams more easily followed.

Figure 1.

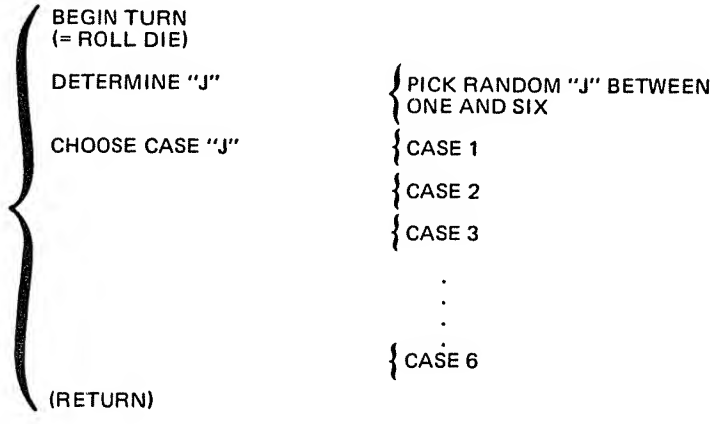
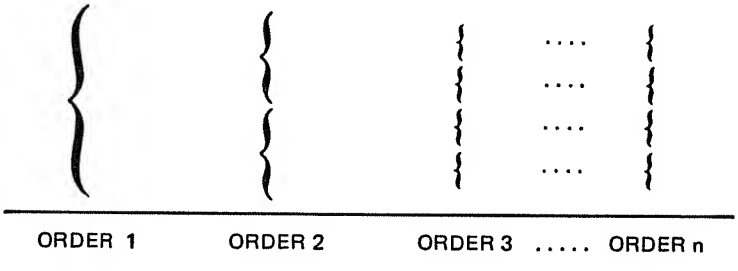


Figure 2.



Dave Higgins replies:

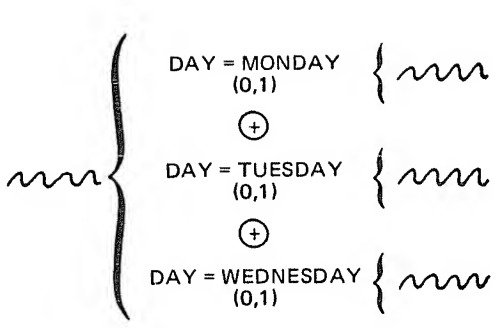
It appears from your letter that you are very interested in using the Warnier-Orr diagramming techniques. I think you will be pleased with the results.

I'd like to comment on the suggestions you made for improving the diagrams. Unlike flowcharts, which have become quite rigid and inflexible in form, the Warnier-Orr diagrams are still in a relative infancy, and do still change occasionally. We here at Langston, Kitch have made some minor modifications to the diagrams in the last year in order to add some capabilities that were previously vague or nonexistent. We are continually evaluating the diagrams, looking for shortcomings or ambiguities, and therefore welcome suggestions along these lines. It is in this light that I considered your suggestions for revising some of the notation.

First of all, with respect to your ideas concerning the representational form of a CASE statement: I think your objection to the use of the ⊕ symbol stems from the fact that there are two primary ways to actually code a CASE structure. One way is with the use of a "computed GOTO or GOSUB." The diagram you show is ideally suited for translation into a computed GOTO, which would look something like listing 1. But I don't think this is a worthwhile change to make to the basic form of the diagrams themselves. The reason is this: although your method works fine for CASE statements that lend themselves to computed GOTO's, there are a whole host of other CASE statements where the use of a computed GOTO is an extreme inconvenience. Take, for example, the CASE of figure 3. It would be inconvenient to have to rig up a computed GOTO to execute this CASE. It is much simpler to code it using a "nested IF" statement, which is the other

[1] *page 9 of this edition.*

Figure 3.



Listing 1.

```
300 REM   CASE STATEMENT
310 REM     DETERMINE CASE "J"
320 LET J=INT(RND(0)*6+1)
330 ON J GOTO 340,380,420,460,500,540
340 REM     CASE 1
350             .
    case 1 process
                .
370 GOTO 570
380 REM     CASE 2
390             .
    case 2 process
                .
410 GOTO 570
                .
    cases 3-6 as above
                .
                .
570 REM   END CASE
```

Listing 2.

```
300 REM   CASE STATEMENT
310 IF D$="MONDAY" THEN 330 ELSE IF D$="TUESDAY" THEN 360
    ELSE IF D$="WEDNESDAY" THEN 400
320 GOTO 440
330 REM   CASE 1: DAY = MONDAY

    monday process

350 GOTO 440
360 REM   CASE 2: DAY = TUESDAY

    tuesday process

390 REM   CASE 3: DAY = WEDNESDAY

    wednesday process

440 REM END CASE
```

Listing 3.

```
300 REM   CASE STATEMENT
310 IF D$="MONDAY" THEN 350
320 IF D$="TUESDAY" THEN 400
330 IF D$="WEDNESDAY" THEN 450
340 GOTO 500
350 REM   CASE 1: DAY = MONDAY
                .
    monday process
                .
390 GOTO 500
400 REM   CASE 2: DAY = TUESDAY
                .
    tuesday process
                .
440 GOTO 500
450 REM   CASE 3: DAY = WEDNESDAY
                .
    wednesday process
                .
500 REM END CASE
```

popular way to code CASE statements. In pseudocode, this CASE is:

```
IF DAY = MONDAY
    THEN MONDAY-ROUTINE
ELSE IF DAY = TUESDAY
    THEN TUESDAY-ROUTINE
ELSE IF DAY = WEDNESDAY
    THEN WEDNESDAY-ROUTINE
```

You can see the natural one-to-one correspondence between the Warnier-Orr diagram and the pseudo-code. This is easily translated to code in listing 2. Listing 3 shows an alternative for those BASICs without the nested IF capability. This is the preferred method for coding a case statement because this method will work for *all* CASE statements, regardless of whether or not the CASE is suited for a computed GOTO. Also, with the computed GOTO, you must be sure that your "J" is restricted to the proper range. This is not to say that you can never use the computed GOTO; just be sure that its use is justified and then be very careful. Personally, I feel it is more trouble than it is worth.

As for the elimination of the brackets with "SKIP" in them: I don't believe that you really want to do this. For instance, in the BUG game published in the October 1977 BYTE[2], no action is taken when a player rolls a "BODY" on the dice but already has a body. This bracket is filled with the notation "SKIP," which indicates that, although the bracket is an essential part of the logic of the diagram, nothing is to be done there. However, in future versions of the game, you might just decide to tell the player that "YOU ALREADY HAVE A BODY" when that condition occurs. If the original diagram is left with the empty brackets intact, you have a fixed and ready place to put that PRINT command. The design is very easy to change and the documentation for the new program is only a matter of erasing one line and replacing it with another.

Also, I don't believe that we need to add the (RETURN) command at the end of the brackets as you suggest. As you state, the return to the next highest level in the diagram is already implied at the end of *each bracket:* therefore adding (RETURN) on each bracket would amount to a lot of "busywork," which would clutter up the diagrams with a lot of unnecessary information.

Again, I'd like to thank you for your suggestions and extend an invitation for all the readers of BYTE to submit their suggestions for improvement of the Warnier-Orr diagrams to either Langston, Kitch and Associates or to me for examination. ■

[2] *page 9 of this edition.*

An Outline Method For Program Design

Jerry Goff

Since I am dedicated to being human, I always try to maximize the returns of an effort while minimizing the effort (a more technical way of saying "Getting the mostest for the leastest"). Therefore, when I read the article by David A Higgins on the Warnier-Orr diagram techniques[1], I thought "AHA!" (or something like that). This is it! The method I now use requires thought, logic and care to get good results. Perhaps the Warnier-Orr method is easier.

I carefully, logically, and thoughtfully constructed a Warnier-Orr diagram of a program. It worked. I then carelessly, illogically, and unthinkingly constructed a Warnier-Orr diagram of a program. It not only bombed, it hung the computer up.

The conclusion is, therefore, obvious. If I already have a method that works every time I use it and I'm familiar with it, why change? Well, so much for the obvious. If it's better, change.

I studied the Warnier-Orr diagram that Mr Higgins included in his article to determine if it was better than my method or if it had something more to offer (I carefully laid aside most of my prejudices), when low and behold, the two methods are the same; only the form is different. Let me sneak in an advantage of my method. It can be stuck in the program as a remark.

I did just that in my version of BUG. You can see that my version (listing 1) of the Warnier-Orr method is in the form of a simple block outline similar to the type forgotten from school. It simply outlines in logical sequence what you want done. Whenever a question needs to be answered, a substatement is generated until all the questions are answered. If nothing happens, simply continue on (just like life).

Try either the Warnier-Orr method or this method. They both work and all you have to lose are ulcers, sleepless nights...

Listing 1: FORTRAN version of BUG program using logical diagramming comments. The entire logic of the program is inserted at the start for documentation purposes. This way you are never further from the documentation than a program listing.

```
0001  FTN4,L
0002        PROGRAM BUG
0003  C
0004  C     ****************************************************************
0005  C     * A) DIMENSION ARRAYS                                          *
0006  C     * B) INITIALIZE PARAMETERS                                     *
0007  C     * C) SET COUNTER FOR COMPUTERS TURN                            *
0008  C     * D) ROLL DIE                                                  *
0009  C     *      DIE=1?                                                  *
0010  C     *         YES   GIVE A BODY                                    *
0011  C     *      DIE=2?                                                  *
0012  C     *         YES                                                  *
0013  C     *           HAVE A BODY?                                       *
0014  C     *             YES   GIVE A NECK                                *
0015  C     *      DIE=3?                                                  *
0016  C     *         YES                                                  *
0017  C     *           HAVE A NECK?                                       *
0018  C     *             YES   GIVE A HEAD                                *
0019  C     *      DIE=4?                                                  *
0020  C     *         YES                                                  *
0021  C     *           HAVE A HEAD?                                       *
0022  C     *             YES                                              *
0023  C     *               FEWER THAN 2 ANTENNAE?                         *
0024  C     *                 YES   GIVE 1 ANTENNA                         *
0025  C     *      DIE=5?                                                  *
0026  C     *         YES                                                  *
0027  C     *           HAVE A BODY?                                       *
0028  C     *             YES   GIVE A TAIL                                *
0029  C     *      DIE=6?                                                  *
0030  C     *         YES                                                  *
0031  C     *           HAVE A BODY?                                       *
0032  C     *             YES                                              *
0033  C     *               FEWER THAN 6 LEGS?                             *
0034  C     *                 YES   GIVE 1 LEG                             *
0035  C     *      ARE THERE 6 LEGS FOR THIS PLAYER?                       *
0036  C     *        YES                                                   *
0037  C     *          ARE THERE 2 ANTENNAE?                               *
0038  C     *            YES                                               *
0039  C     *              IS THERE 1 TAIL?                                *
0040  C     *                YES  SAVE THIS PLAYER AS A WINNER             *
0041  C     *      HAVE BOTH PLAYERS HAD THEIR TURN?                       *
0042  C     *        NO   SET COUNTER FOR PLAYERS TURN & CONTINUE AT D     *
0043  C     *      IS THERE A WINNER?                                      *
0044  C     *        NO   CONTINUE AT C                                    *
0045  C     *        YES   PRINT THE SCORES                                *
0046  C     *          IS THE COMPUTER THE WINNER?                         *
0047  C     *            YES   PRINT THE COMPUTER WINS                     *
0048  C     *            NO    PRINT THE PLAYER WINS                       *
0049  C     *          ARE THEY BOTH WINNERS?                              *
0050  C     *            YES   PRINT IT'S A DRAW                           *
0051  C     * E)  PLAY AGAIN?                                              *
0052  C     *       YES   CONTINUE AT B                                    *
0053  C     *       NO    END PROGRAM                                      *
0054  C     *                                                              *
0055  C     *                                                              ******
0056  C     *                                                              *
0057  C     * HP 21MX COMPUTER        JERRY E. GOFF                        *
0058  C     *                                                              *
0059  C     * BUG ROWS---1=BODY  2=NECK  3=HEAD  4=ANTENNA  5=TAIL  6=LEGS *
0060  C     *                                                              *
0061  C     * BUG COLUMNS---1=COMPUTER   2=PLAYER                          *
0062  C     *                                                              *
0063  C     * WIN(1)=COMPUTER   WIN(2)=PLAYER                              *
0064  C     *                                                              *
0065  C     ****************************************************************
0066  C
0067        DIMENSION BUG(6,2),WIN(2),ITIME(5),IYEAR(1)
0068        DATA L,M,INTEG,REAL /181,66,325,325,0/
```

[1] *page 9 of this edition*

Listing 1, continued:

```
0069  1     DO 3 I=1,2
0070        DO 2 J=1,6
0071        BUG(J,I)=0
0072        WIN(I)=0
0073  2     CONTINUE
0074  3     CONTINUE
0075  C
0076  C     ****************************************
0077  C     * CALL THE TIME FROM THE COMPUTER      *
0078  C     ****************************************
0079  C
0080        ICODE=11
0081        CALL EXEC(ICODE,ITIME,IYEAR)
0082        X=FLOAT(ITIME(1))
0083        X=X/100.0
0084  C
0085  C     ********************************************
0086  C     * START THE GAME,  COMPUTER 1ST            *
0087  C     ********************************************
0088  C
0089  6     DO 100 K=1,2
0090  C
0091  C     ******************
0092  C     * ROLL THE DIE   *
0093  C     ******************
0094  C
0095  7     IX=INT(X*REAL)
0096        IRAND=MOD(M*IX+L,INTEG)
0097        X=(FLOAT(IRAND)+0.5)/REAL
0098        N=INT(10.0*X)
0099           IF (N.GT.6.OR.N.LT.1) GOTO 7
0100  C
0101  C     ************************************************
0102  C     * GO TO THE ADDRESS CALLED BY THE DIE          *
0103  C     ************************************************
0104  C
0105        GO TO (10,20,30,40,50,60),N
0106  10    BUG(1,K)=1
0107        GOTO 70
0108  20       IF (BUG(1,K).EQ.1) BUG(2,K)=1
0109        GOTO 70
0110  30       IF (BUG(2,K).EQ.1) BUG(3,K)=1
0111        GOTO 70
0112  40       IF (BUG(3,K).EQ.1.AND.BUG(4,K).LT.2) BUG(4,K)=BUG(4,K)+1
0113        GOTO 70
0114  50       IF (BUG(1,K).EQ.1) BUG(5,K)=1
0115        GOTO 70
0116  60       IF(BUG(1,K).EQ.1.AND.BUG(6,K).LT.6) BUG(6,K)=BUG(6,K)+1
0117  C
0118  C     ****************************************
0119  C     * CHECK IF THERE IS A WINNER           *
0120  C     ****************************************
0121  C
0122  70       IF(BUG(4,K).EQ.2.AND.BUG(5,K).EQ.1.AND.BUG(6,K).EQ.6) WIN(K)=1
0123  100   CONTINUE
0124  C
0125  C     ****************************************
0126  C     * JUST SOME FORMAT STATEMENTS          *
0127  C     ****************************************
0128  C
0129  75    FORMAT (/,"COMPUTER HAS +")
0130  80    FORMAT ("  PLAYER HAS +")
0131  85    FORMAT (I2,2X,"BODY,",I2,2X,"HEAD,",I2,2X,"NECK,",
0132       *I2,2X,"ANTENNA,",I2,2X,"TAIL,",I2,2X,"LEGS")
0133  C
0134  C     ************************************************************
0135  C     * CHECK FOR A WINNER AFTER BOTH HAVE PLAYED                 *
0136  C     ************************************************************
0137  C
0138           IF (WIN(1).EQ.0.AND.WIN(2).EQ.0) GOTO 6
0139  C
0140  C     ********************************************************************
0141  C     * IF THERE IS A WINNER, WRITE THE SCORES AND WHO WON               *
0142  C     ********************************************************************
0143  C
0144        WRITE (10,75)
0145        WRITE(10,85) BUG(1,1),BUG(2,1),BUG(3,1),BUG(4,1),BUG(5,1),BUG(6,1)
0146        WRITE(10,80)
0147        WRITE(10,85) BUG(1,2),BUG(2,2),BUG(3,2),BUG(4,2),BUG(5,2),BUG(6,2)
0148           IF (WIN(1).EQ.1) WRITE(10,110)
0149  110   FORMAT (" DUE TO INCREDIBLE SKILL,  I WIN")
0150           IF (WIN(2).EQ.1) WRITE(10,120)
0151  120   FORMAT (" WITH ALL YOUR LUCK, YOU MANAGED TO WIN")
0152           IF (WIN(1).EQ.1.AND.WIN(2).EQ.1) WRITE(10,130)
0153  130   FORMAT (" BUT IT'S A DRAW ANYHOW",/)
0154        WRITE (10,140)
0155  C
0156  C     ********************************************************************
0157  C     * PROGRAM EXIT (THIS IS HANDY FOR STOPPING THE PROGRAM)            *
0158  C     ********************************************************************
0159  C
0160  140   FORMAT(" WANT TO PLAY AGAIN?  1=YES,  2=NO +")
0161        READ(10,*) ANS
0162           IF (ANS.EQ.1) GOTO 1
0163        END
```

Common Mistakes

Using Warnier-Orr Diagrams

David A Higgins

In my opinion, one of the best program and system design methods is the Warnier-Orr structured systems design approach, which I described previously ("Structured Program Design," page 146,[1] October 1977 BYTE; "Structured Programming with Warnier-Orr Diagrams," page 104,[2] December 1977 and page 122,[3] January 1978 BYTE). This article is being presented because of the interest expressed in this subject, and because a lot of people will be trying these techniques for the first time. Newcomers to this methodology often have many questions about their work, and want to know whether or not what they are doing is correct. The purpose of this article is to outline a few of the more common mistakes that beginners make when using this technique.

Editorial Note. . .

Since publishing David Higgins' first two articles on Warnier-Orr diagramming techniques, we have received a number of letters from people expressing the message (paraphrased) "if I have this or that self-documenting structured programming language, why should I use Warnier-Orr techniques? After all, if a program in my language is logically equivalent to the Warnier-Orr structure, and it is directly executable, I see no need for an extra layer of documentation."

A very real answer to this objection is that it is correct. There is no point to using Warnier-Orr techniques if you properly use a language such as PASCAL which, having structured programming constructs built in, allows long descriptive names for variables and procedures, and as a result can support self-documenting code.

But most currently used languages in personal computing do not easily support self-documenting code and modern concepts of structured programming. The usefulness of the Warnier-Orr methodology is that it provides a disciplined way of imposing such structure on a language such as BASIC, FORTRAN or assembly language. In effect, the Warnier-Orr discipline is a programming language which is intended for hand translation into one of the existing unstructured languages. . . **Carl Helmers, Editorial Director BYTE Publications**

Philosophical Errors

Many first time users of the Warnier-Orr diagrams tend to make mistakes which are so similar that they are worth examining. The biggest and most common mistakes tend to be a direct result of what we can call philosophical errors; not really a misuse of the techniques so much as a misunderstanding of the techniques. The most common error stems from the fact that many computer programmers tend to be obsessed with the desire to write some kind of code at the very beginning of the design process. This problem usually manifests itself in any or all of the following three ways:

- Trying to code the program while designing it (called the design-a-little, code-a-little approach).
- Relying too heavily on language restrictions and considerations while doing logical design.
- Skipping the design phase altogether because:
  a) the program is "too easy" or
  b) the programmer is "too smart."

Any of the above practices will destroy most if not all of the effectiveness of the Warnier-Orr methodology *[or any other structured programming methodology for that matter. . . CH]*. It will certainly cause you to waste a great deal of time.

If you try to use the first technique, the design-a-little, code-a-little approach, you will probably be in for quite a bit of erasing or retyping when you have to change the design because you coded yourself into a corner that you can't design your way out of. Your program will tend to be twice as long as it should have been and half as efficient. You will probably be in for a lot of debugging runs while trying to put back into the code everything that you left out when you changed the design. As you can see, this technique just naturally generates problems.

The second technique described above is a common mistake that veteran programmers almost always seem to make: relying too heavily on the program language they will be using while doing the program design. Consider the two examples of program

[1] *page 9 of this edition*
[2] *page 14 of this edition*
[3] *page 19 of this edition*

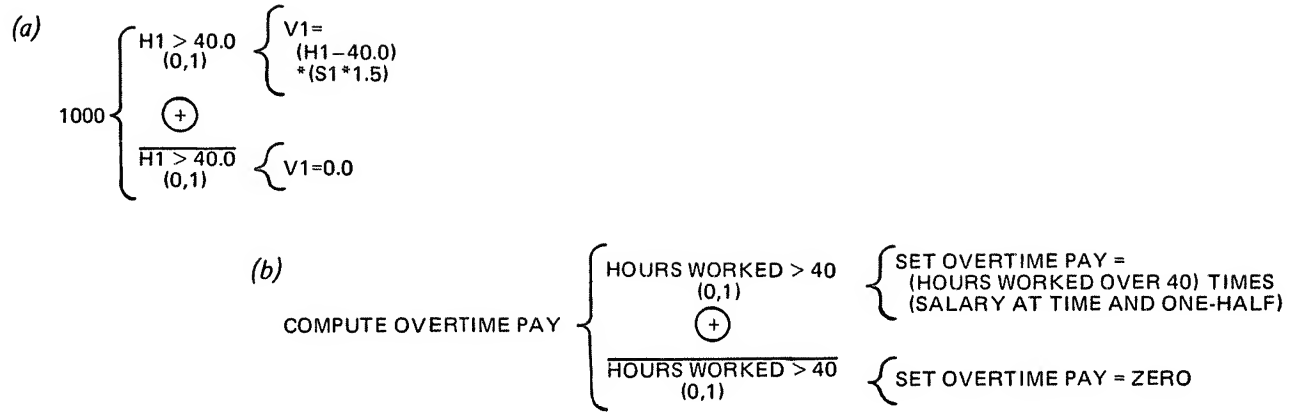


Figure 1: DOs and DON'Ts of Warnier-Orr diagramming. Figure 1a looks like actual program code and should not be used when trying to logically design a program. Figure 1b shows the correct method. The entire diagram contains only logical statements which could be coded into any computer language.

designs shown in figure 1. Both figures 1a and 1b are diagrams of the same process: computation of overtime wages. The diagram in figure 1a however seems to be the type that veteran programmers will almost always try to draw. Note its heavy stress on the language aspect of the function. It almost looks like part of a BASIC program cut out and pasted on a diagram. Contrast that diagram with the one of figure 1b which correctly details the logical process being performed. You can see that if figure 1a was the only documentation for this particular procedure, you would probably not be able to tell what that piece of code was supposed to be doing. You might have some idea because this program seems to have semimeaningful field names from which you might deduce some purpose. All we can tell for sure from figure 1a is that some part of the program is going to crunch a couple of numbers. What numbers it is going to crunch and just what for are anyone's guess. On the other hand, it is impossible to misunderstand what the process diagrammed in figure 1b is doing. It is very easy to read and comprehend because it shows the logical side of the procedure.

This stress of the logical over the physical while designing with the Warnier-Orr diagrams is essential to their correct usage. Designing as in figure 1a serves absolutely no purpose as far as understanding the process that is being described and is essentially worthless as far as documentation is concerned. Even though you might be able to tell what that diagram does the day you draw it, you probably won't be able to understand it in six months. Someone else who wants to use your documentation might never understand it.

As long as we're on the subject of documentation, I might mention that through the development period of this technique, many people were concerned that the diagrams might become too far removed from the actual code, which would render them useless as effective documentation. They worried that since the diagrams depicted the logical side of the problem, they had little or no relevance to the physical (real world) side. Those fears were easily put aside with two diagramming and coding conventions, as follows:

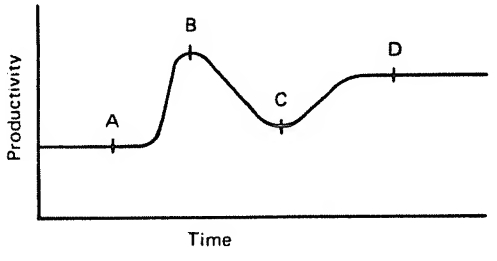


Figure 2: Typical productivity curve of programmer being introduced to Warnier-Orr diagram methodology.

- Physical mileposts on the Warnier-Orr diagrams.
- Logical symbol tables in the programs.

Thus, when we actually wrote code that looked like that of figure 1a, we would tie it to the logical figure 1b by adding the following to the diagram.

This would be included in the program itself by using comment statements:

```
      .
      .
      .
1000 REM     COMPUTE OVERTIME PAY
1001 REM        HFLD    = HOURS WORKED
1002 REM        OVTFLD  = OVERTIME PAY
1003 REM        SALFLD  = SALARY
      .
      .
      .
```

This allows us to have a very clear and concise, one to one mapping between the logical diagram and the physical code. References between the two diagrams are quite easy. If, for instance, you want to know what a particular section of code is supposed to be doing, you need only to look it up on the logical diagram. Similarly, if you want to find out which part of the program is carrying out a particular logical function, you have the location information at your fingertips. This is excellent documentation in the event that you or someone else might someday want to make a modification to your code.

The third common philosophical error, that of skipping the design phase altogether, is a real problem to most newcomers. In fact, if you look at a typical productivity curve for a programmer who is introduced to the Warnier-Orr diagrams, it generally looks something like the curve in figure 2.

A currently productive programmer producing work at a constant rate up until the time the Warnier-Orr techniques are introduced (point A), will typically show an initial burst of very high productivity (point B). This is usually followed by a slump (point C) where the programmer sinks back to or just above his previous level of work. Eventually, he will climb back up to a new, higher level of work (point D), where he will usually stay. This peculiar slump at point C seems to be primarily due to the fact that since the programmer has begun to feel comfortable with the new technique and has had some initial success with it, he begins to feel confident enough to try to do the work without doing the diagrams first. He soon realizes that the quality of his work has dropped off and starts to do the diagrams once again, this time for good, and his work level rises up to a new, higher level that will remain fairly constant.

Apparently, the only way to get new people to avoid this temptation is to forewarn them that it does tend to happen, so that if and when they find themselves on the downhill side of the productivity curve, they can recognize the trap in time to escape the worst of it.

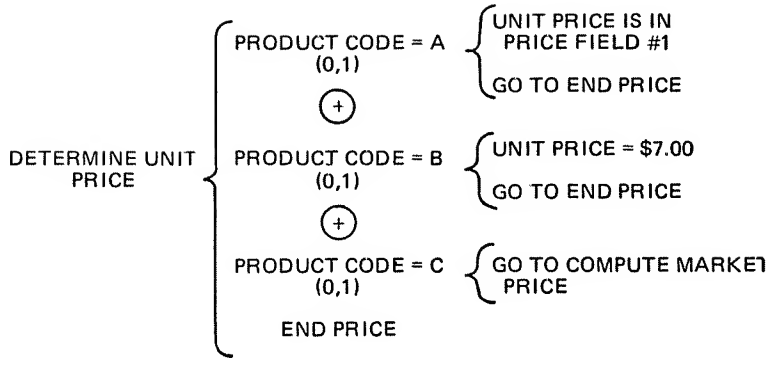


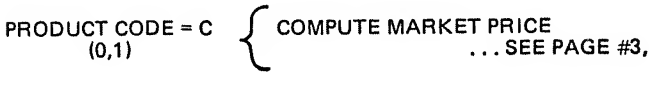
Figure 3: Example case statements making use of logically illegal GOTO statements. When a set of statements is finished the diagram will logically fall through all of the other exclusive ORs, ⊕, and arrive at the END PRICE section. Thus no GOTO need be shown.

So much for the philosophical errors. There are also a few common technical errors that people make, and we'll look at those next.

Technical Errors

For a lot of people who are just starting to program and may be unfamiliar with structured programming techniques, some of the diagramming methods may seem to be a bit uncomfortable. One of the most often seen technical errors is the attempted use of a GOTO statement on the diagram. The case statement shown in figure 3 illustrates this problem.

Two of the occurrences of the GOTOs in figure 3 are incorrect and the other is ambiguous. The GOTOs in "PRODUCT CODE = A" and in "PRODUCT CODE = B" are unnecessary and incorrect. The default logical linkages will see to it that the appropriate steps are executed. The GOTO at "PRODUCT CODE = C" is unclear. If it is supposed to mean that we are to cease execution of this process and jump to the procedure "COMPUTE MARKET PRICE" to begin processing, then its usage is incorrect. If on the other hand it means that "COMPUTE MARKET PRICE" is a common utility routine and is described elsewhere in the system, then the GOTO is misleading. Instead, we should have written:



if the process was expanded on a different page of the diagram; or something like the words ". . .SEE ABOVE" or ". . .SEE BELOW" if that process appears elsewhere on the same page. The GOTO is a physical entity to be used at execution and is not a logical relationship, so it does not belong on

Figure 4: Example of a case statement with processes that are mutually exclusive and mutually independent.

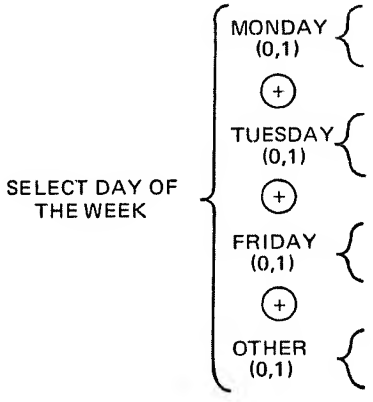


Figure 5: When a case statement has mutually independent and mutually exclusive statements, the statements may be rearranged into any order without changing the logic of the diagram.

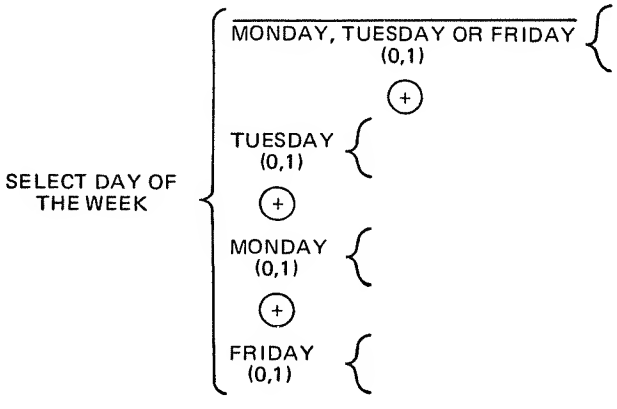
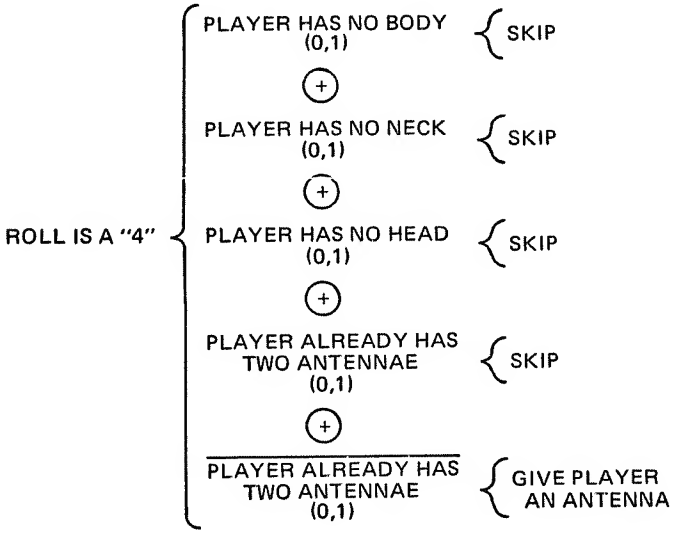


Figure 6: Although this is a working Warnier-Orr diagram, the case statements are not mutually independent.




```
IF 'player has no body'
   THEN . . .
ELSE IF 'player has no neck'
   THEN . . .
ELSE IF 'player has no head'
   THEN . . .
ELSE IF 'player has two antennae'
   THEN . . .
ELSE 'give player one antenna'
```

Listing 1: Typical if-then-else structure for Warnier-Orr diagram of figure 6.

a logical Warnier-Orr diagram.

Another common technical mistake is one that is a little harder to catch, and is one that even professionals with this technique will make if they aren't careful. Consider the case statement shown in figure 4.

Note that in this case statement, not only are the processes outlined mutually exclusive (only one of the cases is true), but they are also mutually independent. That is, their order within the case statement does not matter. It would be just as correct for me to have written the diagram as shown in figure 5.

In an earlier article "Structured Program Design" (Oct 77 BYTE)[4], the game of BUG was outlined. In the game, a die is rolled for each player and each number of the die corresponds to a part of the bug's body; the player finishing his bug first wins the game. If a player rolls a 4 for instance, he is entitled to one antenna. But he must have already acquired a body, a neck and a head in that order before he can receive an antenna. He needs a total of two antennae if he is to complete a bug.

Many people would try to code that process as a case statement as in figure 6. The process in figure 6 certainly looks correct, and indeed, if you code it as a case statement, as in listing 1, it will even run correctly.

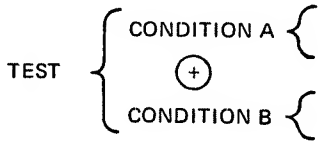
However, this process is not a case statement. It is more properly called a pseudo-case statement, because each of its cases is mutually dependent. The cases cannot be reordered within the statement without destroying its logic. Notice that rearrangement of the case statement diagram as shown in figure 7 does not work at all. This arrangement will give the player an antenna anytime a four is rolled, until he has two antennae, regardless of whether or not he already has a body, a neck or a head. A more correct logical interpretation of the case structure we want is shown in figure 8.

You might also notice that since the bug must have a body before it can have a neck (and a neck before it can have a head) if we merely check for the presence of the head, we will be indirectly checking for the neck and the body, so that figure 9 is an equivalent

[4] *page 9 of this edition*

structure.

Another common technical error is the misuse or lack of use of the (0,1) notation in conjunction with the exclusive OR, ⊕. Many times, people will simply write:



By this they often imply the (0,1) notation with the use of the symbol ⊕ alone. Actually, this is not incorrect; in fact, for most people familiar with the diagrams, this notation seems to be just as clear. But for users not quite familiar with the Warnier-Orr diagrams it is probably best to go ahead and include the (0,1).

To conclude, I'll reiterate a point made in an earlier article: Understanding a Warnier-Orr diagram is very easy; creating one from scratch is much harder than it looks.■

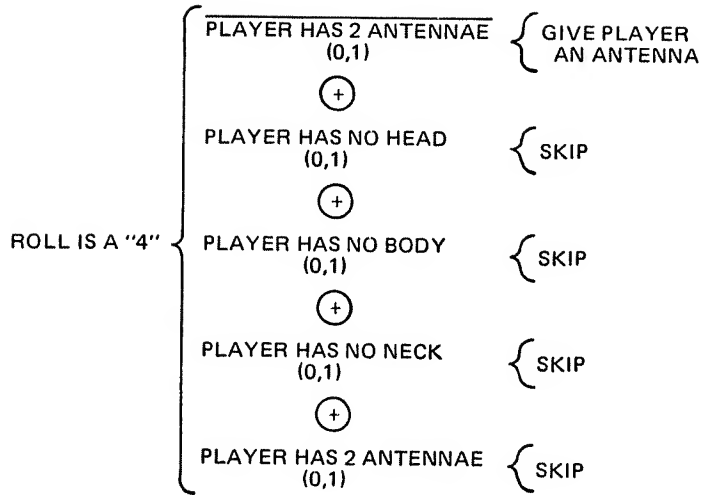
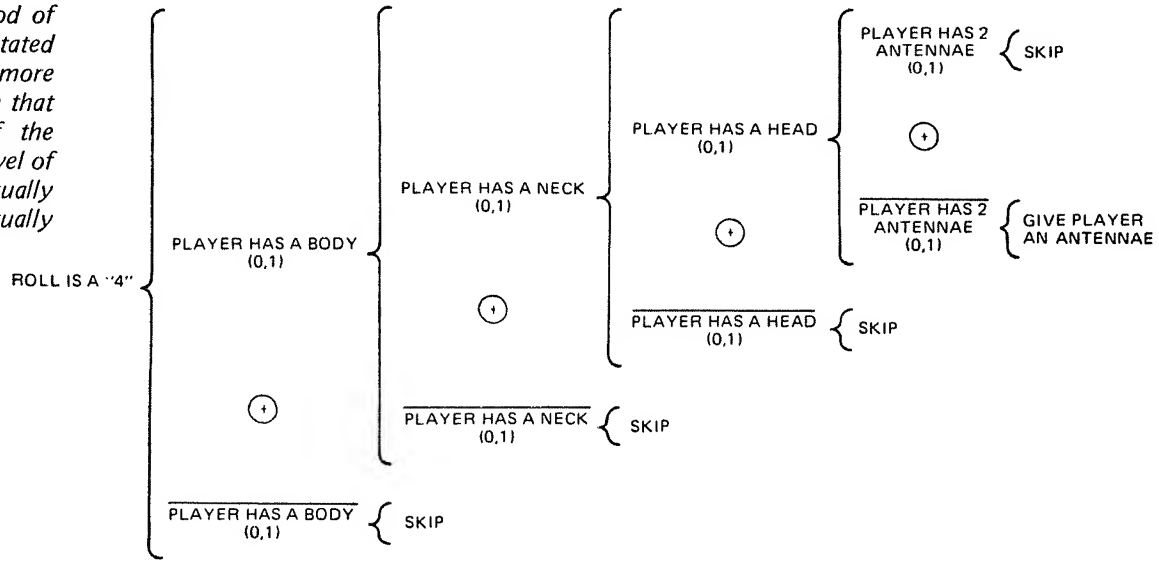


Figure 7: When the statements in figure 6 are rearranged as shown, it can be seen that the program fails to work as desired.

Figure 8: This method of approaching the stated "bug" problem is more logically correct than that of figure 6. All of the statements at each level of the diagram are mutually exclusive and mutually independent.



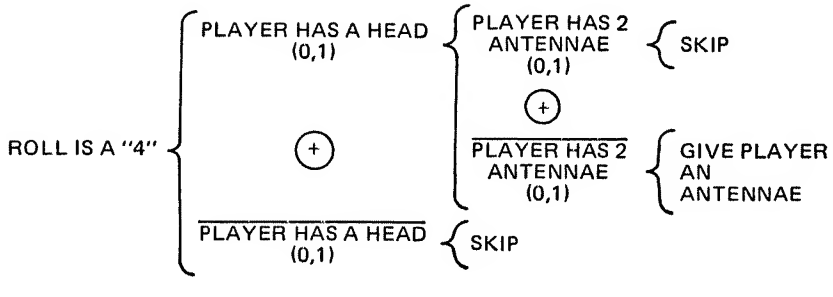


Figure 9: Since a bug must have a head in order to have an antennae, and a body and neck to have a head, the search process can be shortened by just checking for the presence of a head.

Top-Down Modular Programming

Albert D Hearn

If you have done some programming, you know that it's one of the most enjoyable and satisfying parts of personal computer use. The very thought that the vast power in the small system's processor is limited only by the program that you write for it is tremendously exciting.

If you are new to the computer game, the programs you have written up to now have probably been relatively small and uncomplicated, but you have developed a lot of experience and confidence from them. Most likely you haven't used any particular technique in designing and writing your programs: you have probably approached program design in an informal way and relied upon your good senses to guide you in this unfamiliar task. You have probably also gained an understanding of the full capabilities of the instruction set and some of the little tricks (yes, ADDing a binary number to itself really does result in a left shift of one bit) which can be so useful. You are also capable of writing IO routines to do about any kind of data transfer you want.

So now you are ready to do a program which does something really useful. The program you have in mind is going to be larger and more complicated than those you have done previously. While you might not expect this, your previous informal methods of designing and coding might possibly be inadequate and could cause you much grief if you attempt to use them on a larger program.

Hopefully, I can help you prevent these kinds of difficulties by showing you in this article an easy to use method of designing and structuring larger programs which can greatly simplify your personal efforts, regardless of complexity.

The Concept

Someone once said, "To solve a complex problem, simply break it down into a number of less complex pieces, then proceed to solve it one piece at a time." This approach has been used for many years in the design and building of electronic equipment. It results in a "building block," or "modular" construction, where each block or module does some distinct part of the total function of the equipment. For instance, think of the last time you saw a diagram of a radio receiver. It was probably in the form of a set of separate blocks representing the RF amplifier, mixer, IF amplifier, and so on. The blocks were all connected with flow lines showing the sequence in which each equipment module processed a signal coming from the antenna. The diagram enabled the reader to understand the function of the radio one module at a time, in relation to the whole radio.

So how does the idea of using building blocks and solving problems piecemeal relate to the programming of personal computers? The answer is that these same ideas are very applicable to programming and have been in use in commercial programming for a number of years. There is no reason that good use of them can't be made in the amateur computer hobby also.

Top-down Design

Top-down design of microprocessor programs requires that you first have a clear notion about what it is that you want the program to do. You should ask yourself questions like, "What function do I want performed?", "What input information is available?", and "What output information

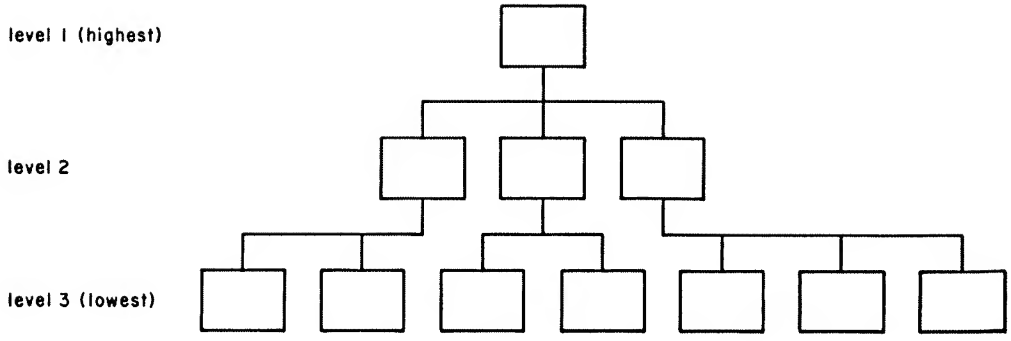


Figure 1: A basic top-down design diagram is a structure like this. The number of levels may vary, and the number of boxes may vary, but the basic idea is given by this prototype.

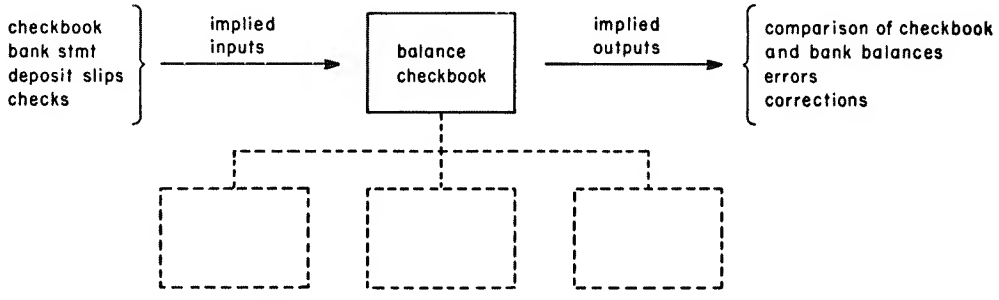


Figure 2: The first level of design is the act of saying "I want a program to do thus and so." Here "thus and so" is defined to mean checkbook balancing.

or action do I expect?" When you can answer these questions, you've actually completed the highest level of design.

The basic principle of top-down design procedure says that you start at a very high level of function definition and then progressively expand that function into more and more detail until you're at a low enough level to begin coding your program. Actually, this is a very natural way to design solutions to any problem, but, for some reason, this method was very slowly applied to programming. The top-down method is different from bottom-up, where the concern is for coding and details before a real program design has been done. Bottom-up methods work on the "how" aspects of the program before the "what" aspects. An analogy of this method would be the building of a house, using no structural plans, by first laying down a convenient foundation and then gradually adding wood and stone until some desirable structure has evolved.

Let's take an example of a function that could be performed on a microprocessor system for the purpose of illustrating the technique of modular, top-down program design. The function, monthly checkbook balancing, was selected because it is a process that is familiar to most of us and it contains all of the elements which make it a good example.

In order to design what you want the program to do, begin by drawing a multi-level design diagram like the one shown in figure 1. The diagram will describe what the program does at a number of different levels of detail, starting with the highest level which is a single block describing the overall function. The next lower level of blocks breaks the higher level function into a number of more detailed subfunctions. The next level takes those blocks and breaks them into even greater detail, and so on. An important point to remember is that the total function of the program is represented at each level.

Figure 2 illustrates the first steps in the top-down design of your checkbook balancing program. The first block simply states that the program will balance your checkbook. There are no details in that block and it certainly doesn't invite coding at this point in the design. For input, you know that you will have your checkbook entries, monthly statement from the bank, deposit slips and cancelled checks. The output you want is a comparison of your checkbook balance (adjusted for recent deposits, service charges and outstanding checks) and the balance shown on the bank statement. You also want to know where any errors were made and what corrections are required.

The second level of design, shown in figure 3, breaks the first level block into three major subfunctions. Although this subdivision could have been done differently in terms of the content of the second level blocks, the sum total of those functions always adds up to the entire function of the program. The idea is that you start the

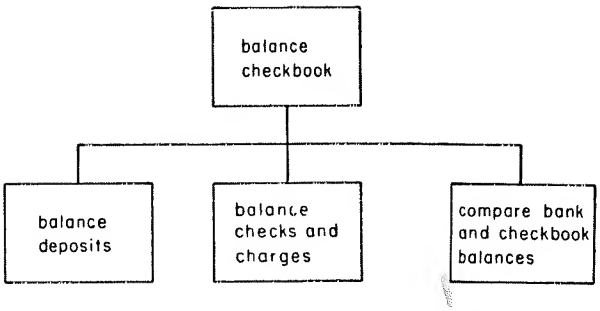


Figure 3: Once the first level of design has been determined, the next level is specified by breaking up the task into parts which are fundamentally independent of one another. Here, checkbook balancing is viewed as three separate modules of function.

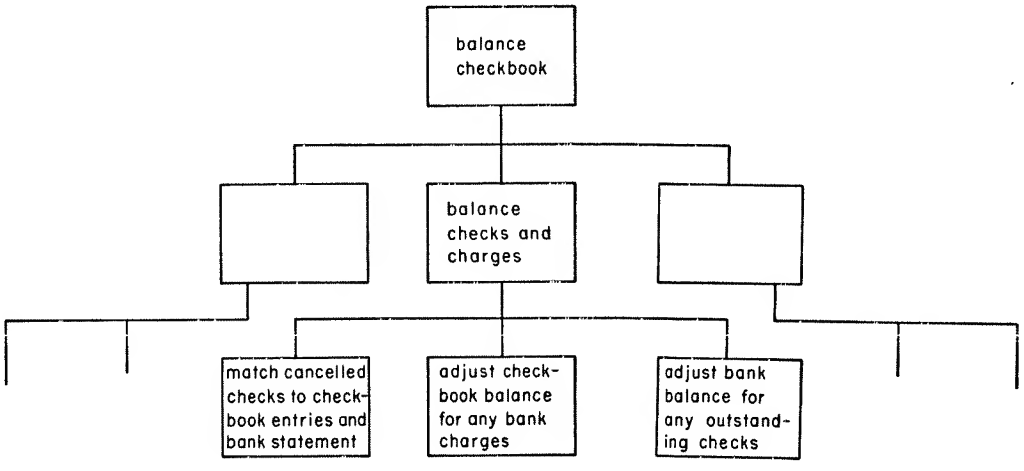


Figure 4: Carrying the process one step further, the next level is shown here for one of the branches of the structure of the programs.

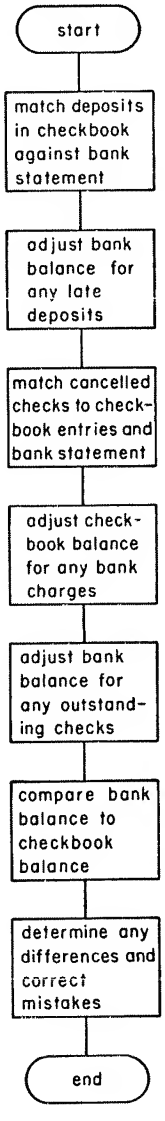


Figure 5: After the modular structure of the application is determined in a hierarchy such as those exemplified in figures 1 to 4, then attention can be given to sequencing of functions. This flowchart shows general level sequencing of the checkbook balancing application.

process slowly and don't attempt to develop too much detail too soon. Keep the number of subfunctions small, five or fewer, under each function block. Don't worry about the order in which these subfunctions will be performed in your program. Remember, you're only concerned at this point about what is to be done, not how it is to be done.

Next, take the design to the next lower level by further subdividing each of the second level blocks. Figure 4 illustrates a portion of this step. Just make sure that each subblock represents a complete subfunction and that the subfunctions at any level are equivalent to the program function.

You might ask at this point, "How many levels must I go through?", or "How do I know when to stop?" There is no precise answer to these questions, although the following guidelines should help. In general, you will find that you should stop when the lowest level of functions is so simple that you can easily write a program module to do each one. A module should be considered to have about 50 program instructions, or less. Experience will help you to know when you have reached this point. Also, you will find that the more complex the program, the more design levels you will need; generally, about three or four levels will be sufficient.

Another method of determining if you've carried the design to a low enough level comes about almost automatically. If you are attempting to complete one of the lower levels and you find that the order of subfunction execution is becoming difficult to ignore, then you've probably gone far enough. Also, if you find that it is becoming necessary to show that program branching or decision making is required (top-down design diagrams should show no decision logic), then you know that you have about the right level of design. You are now ready to start thinking about the how of your program.

Modular Construction.

If you try to make each block at the lowest level of your design diagram into a module, you might determine that some blocks are simple and can be combined into fewer modules. On the other hand, there will probably be blocks which would result in modules larger than the minimum size of 50 instructions we have established. In this case, take the blocks through one or

more additional levels of design.

Now decide what sequence the functions should be performed in. Begin drawing a flowchart showing the required sequence. Will each function be performed for each pass through the program? If not, add decision blocks showing the conditions under which each such function is executed. Also add any function blocks which may be necessary to initialize data, clear tables, IO data, etc.

Figure 5 shows a sequence of functions which results from the design of your example checkbook balancing program. Actually, the functions shown are probably too high level for this step, but for the sake of illustration, the diagram should make the point.

At this time, I would recommend that you consider making use of a special program structure called an executive routine, which offers some significant advantages. The executive is the main routine in the program and primarily contains calls to the function modules which do all the processing duties. It makes all decisions about the sequence of execution. It also contains the starting and ending points of the program. The objective of the executive is to concentrate most of the decision logic and common function of the program into a separate routine which becomes another program module.

In this way, the function modules need not, and should not, make sequencing decisions. They should never directly pass control to another function module. This should be done only through the executive. A function module's only responsibility is to be given control by the executive, do its assigned job, and then return control back to the executive. Function modules are written in the form of subroutines using the call and return facilities of the programming language being used. They should also contain a generous sprinkling of comment statements to insure a high degree of understandability, as well as a well-defined IO interface to the outside world and the rest of the program.

Figure 6 illustrates the final step in the modular, top-down design of your checkbook balancing program. You have added an executive routine and some necessary housekeeping routines. You could begin coding the program from this flowchart by first writing the executive and the associated subroutine calls for each of the processing modules. By writing dummy subroutines which simply return control when they are called, you can test your executive for correct operation without the need for the real processing modules.

The next step, of course, is writing the processing routines. This is simplified by the design approach described in this article because it allows you to work on each routine as a separate unit which can be written and tested independently of all other routines in the program. When all routines are completed, they simply plug into the executive to form a total program. Later, if you want to change the sequence of execution, add or delete functions, it can be simply a matter of manipulating modular routines. ■

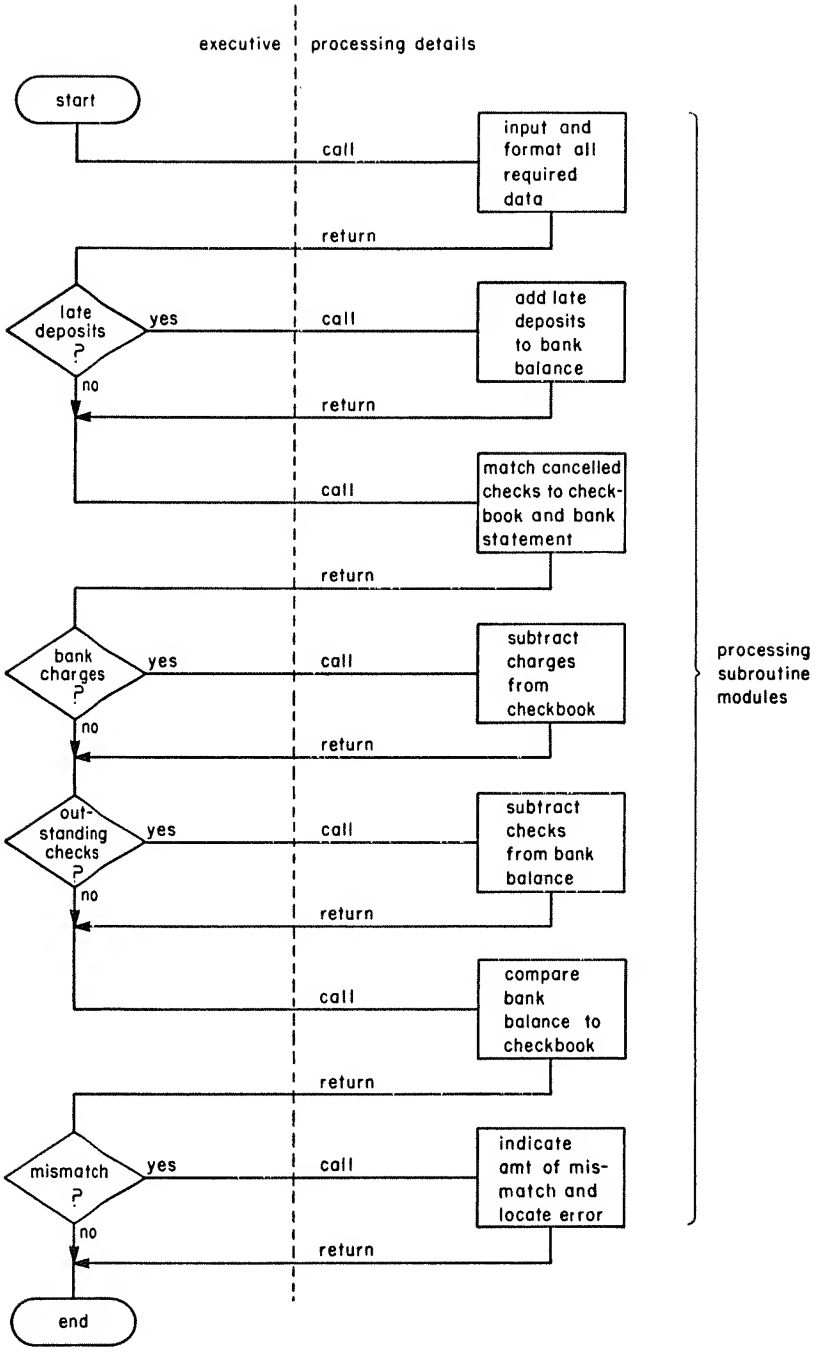


Figure 6: While the sequencing of the diagram shown in figure 5 is adequate, it is often useful to explicitly partition all sequencing of execution in a separate module called the "executive" for the application. This flowchart shows a simple example of such an executive program which sequences the major operations of the application.

Some Words About Program Structure

Albert D Hearn

Microprocessor programming, at this point in time, is a black art. Once you have learned the basic instruction set, you're on your own. Some people get the knack of this mysterious task fairly quickly, and some do not. Those who do well seem to have developed some sort of system for going about it. The point is that an organized, systematic approach is required if there is any hope for continued programming success. The purpose of this article is to describe to you one such method which has become very popular with programmers of all types, using all kinds of computers from micros to the giants.

Concept

What we're looking for is simplicity in the writing of programs. This is usually achieved if the program can be reduced to a collection of basic components which fit together in very well-defined ways. This is the concept behind structured programming.

Any program can be considered to have only two basic building blocks. One is the process block shown in figure 1. It simply performs some defined function, or process. It might represent a simple function requiring only a few, maybe only one, instructions in the program, or a much larger function requiring many instructions. Whatever it does, it has one input and one output.

The second basic block is the decision block shown in figure 2. This elementary capability of any computer is that which gives it all its power and flexibility. It is the ability to alter the path taken by the program based upon the value of some parameter or condition which can be tested by certain instruction types. For example, two numbers can be compared and a test for equality used to decide which of two program paths will be taken as a result.

These two fundamental building blocks will now be used in the construction of a set of basic program structures with which any other program can be built. The three general structures are called *sequence*, *if-then-else* and *loop*. Variations of these will be examined, as well as combinations which can be used to build more complex functions.

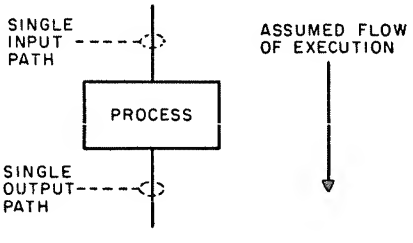


Figure 1: The process block is the "black box" of programming: It is entered by a single input path, does some arbitrary operations upon data, and is exited by a single output path. The "arbitrary operations" can be as simple as one step in an arithmetic calculation, or as complex as a compilation of a program — it all depends on the point of view taken.

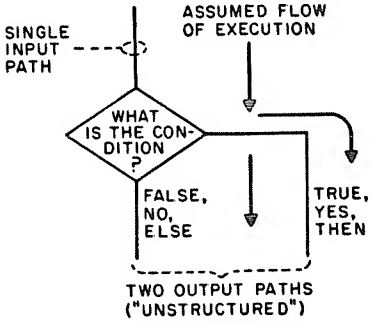


Figure 2: The decision block is a simpler concept than the process block, in the sense that the amount of computation required rarely approaches the generality of an "arbitrary process." A decision block has one input and, depending upon a binary condition, takes one of two output paths. In this figure, the names "true," "then" and "yes" denote one possible path; the names "false," "else" and "no" describe the other possible path. In programming languages, the "then" or "else" terminology for the two paths is frequently built into the language design; the other terms are frequently seen in flowchart representations of programs.

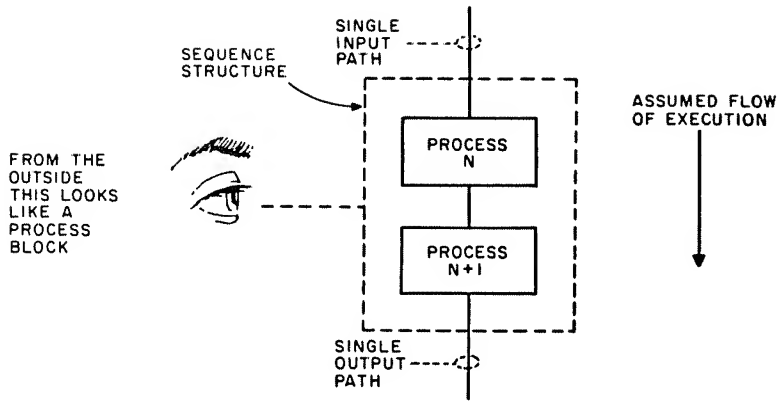


Figure 3: The sequence *structure is the simplest programming structure. It can be viewed from the outside as the equivalent of a process block, but upon close examination it is found to contain one or more process blocks.*

Basic Structures

The simplest of the program structures, shown in figure 3, is the *sequence* structure, which is composed of one or more process blocks strung together serially. Like the process block from which it is built, the *sequence* structure has only one input path and one output path. In fact, you will soon see that one of the rules that we want all structures to conform to is that they have a single input path and output path. Furthermore, an entire program, which can be represented by one large process block, should also conform to this rule.

The next structure is the *if-then-else* structure, shown in figure 4. It consists of a decision block and two process blocks. Only one of the process blocks is executed for any single pass through the structure. The result of the test or comparison repre-

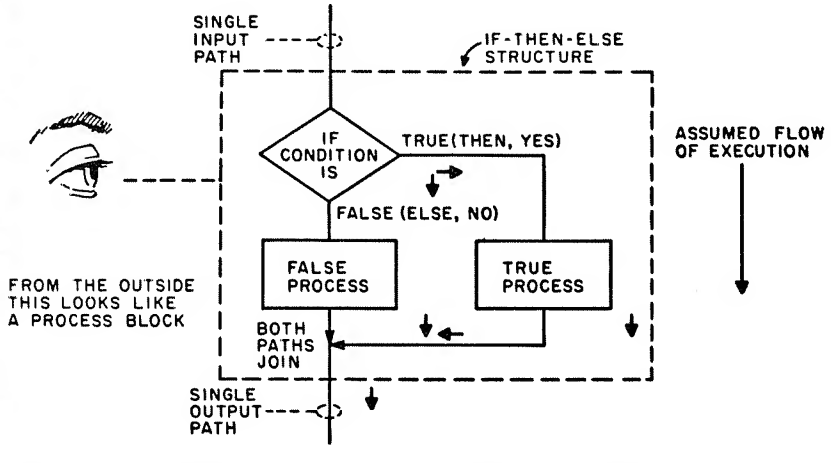
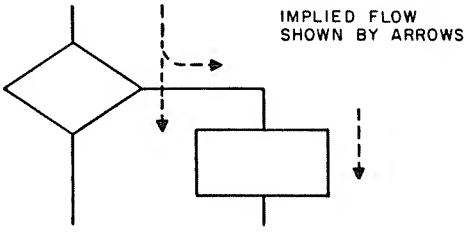


Figure 4: The if-then-else *structure is composed of a decision block and two process blocks. The process blocks may themselves be viewed as any form of structure with a single input and a single output path, and thus might in fact be* sequence *structures,* if-then-else *structures, etc.*

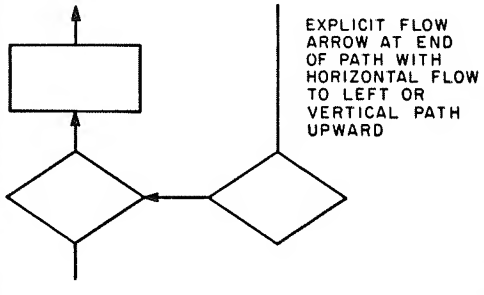
Editor's Note: BYTE Flowchart Flow Conventions

As an "ideal" standard, flowcharts in BYTE use a direction of flow convention as follows:

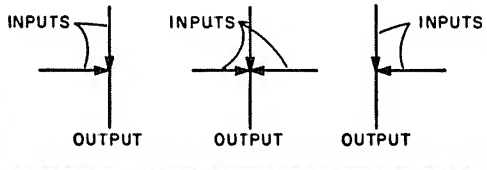
Default flow: *Vertical flow is from the top of a diagram toward the bottom, and horizontal flow is from the left of a diagram towards the right, unless explicit flow is used. Thus:*



Explicit flow: *Vertical flow upward, or horizontal flow leftward in a drawing, is shown with an explicit arrow at the end of the flow path, thus:*



Merged flow: *When two or three paths of flow merge the two or three inputs to the joint path have arrows noted:*



sented by the decision block determines which process block is chosen. Notice that regardless of which path is taken there is one common exit path from the *if-then-else* structure. This is required to maintain our single exit philosophy.

An *if-then-else* structure does exactly what it says: *if* a condition is true, *then* take a specified action, *else* take a specified alternate action. However, there are times when only one action is required in only one of the paths. No action is necessary in the other path. In an actual flow diagram, this is of course shown by drawing a flow line in place of one or the other process block of the *if-then-else* structure since the most trivial process is simply going to the next process without doing anything. Note however that only one of the process blocks can be made up of this simplest case of "do nothing" since if both process blocks were eliminated from the *if-then-else* structure, the net effect would be to "do nothing" all the time whether or not the condition was true or false.

The *if* part of an *if-then-else* structure is simply any program instruction which can perform a test and take one of two paths depending upon the outcome. In an assembly language, this is usually a conditional jump or a branch instruction based upon the outcome of some comparison, arithmetic operation or other operation which affects processor status flags used in such branches and jumps. The branching instruction specifies the destination address of the beginning of one path, whether it is the *then* or the *else* leg is arbitrarily defined, and the next sequential instruction is assumed to begin the opposite path.

Some higher level languages like BASIC have ready-made *if-then-else* instructions. BASIC has IF and THEN; ELSE is implied. The following shows how an *if-then-else* would look in BASIC:

```
1  IF X=Y THEN 10
.  .......
.  .......   } FALSE PART
.  .......
.  GOTO 15
10 .......
.  .......   } TRUE PART
.  .......
15 END
```

In this example, the *else* code immediately follows the IF instruction. The GOTO 15 ends the *else* path and causes the program to branch to the common exit point at line 15. The *then* path starts at line 10 and ends at line 15. [*BASIC is considered to be an "unstructured" language because of the need for an explicit GOTO following the "false part" of an IF-THEN-ELSE construction.*]

If you use assembly language in your programming, and your assembler has a macroinstruction capability, then you can write your own *if-then-else* macros. It is beyond the scope of this article to describe how this is done, but it isn't very difficult.

If you use assembly language and don't have facilities for writing macros, then you can simulate the function of the macroassembler in order to gain the advantages of structured programming. Simply sit down and write yourself a set of standard *if-then-else* structures. Take the five or six most common decision types (equal, not equal, zero, greater than, etc) and write skeleton programs for each. Leave blanks for the actual condition to be tested, and leave space for the actual code which will perform the *then* and *else* functions. Later, when you need an *if-then-else* while writing a program, you can draw upon your set of prewritten structures. Not only does this eliminate your having to invent similar program sequences over and over again, but it also prevents many bugs and greatly eases the effort you have to put into program writing.

The last basic structure is the *loop*, which provides a means of repeating a sequence of instructions until some stop condition is found to exist. There are two kinds of loop structures: *do-until* and *do-while*.

A *do-until* structure, shown in figure 5, performs the function in the process block at least once. After that, a test is done to determine if the condition for stopping the process looping has been found true. As long as the condition is not true, the looping continues. When it becomes true the looping ends and the exit path is taken. This type of structure can be used, for example, when you need to search a table of values, looking for a particular value. If you know that the table will always contain a matching entry, the program routine need not be more complicated by logic to detect end-of-table before a matching value is found. Notice that the first table entry is always examined before the decision is made to continue (this is because the ending condition decision is based upon the value of that entry).

The second type of loop is the *do-while*, shown in figure 6. The difference between this and the *do-until* structure is that the test is done before the process block is executed. In many cases there is not a lot of significance to this difference because both types of structures can do the same

jobs.

In specific situations you will find that one form will usually be better suited or more convenient than the other. The primary difference to remember is that the *do-until* form always executes the process block at least once whether or not the *until* condition is true, and that the *do-while* may not execute the process at all if the *while* condition is false at the time of the first test. Experience will best teach you which to use in the various situations.

A variation of the loop structures of either form might be considered, the endless loop or *do-forever*. This form of loop occurs when the *while* or *until* condition is never changed to allow execution of the output path of the structure. Intentional endless loops are occasionally used, as in the low level programming trick of hanging up execution in a tight loop to flag errors, or the quite legitimate endless loops which form the outer level of control of a typical executive or monitor program. But for most programming purposes, an endless loop is a bug or error in the program.

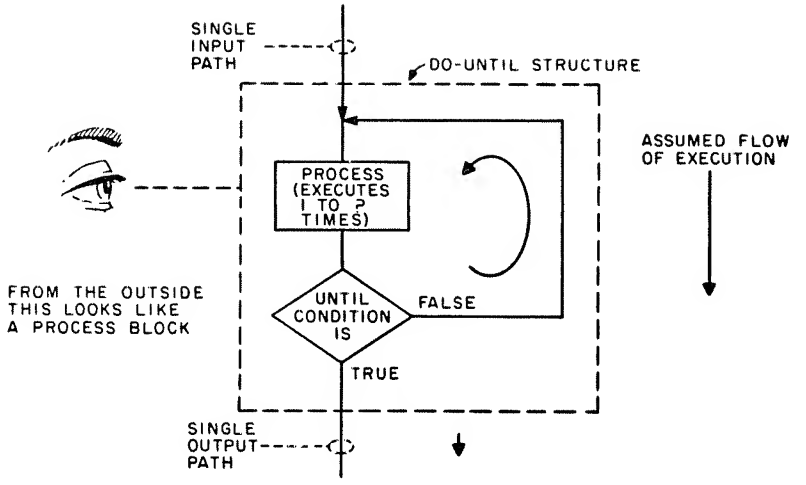


Figure 5: The do-until *structure is a looping form whose purpose is to execute a given process block at least once. After executing the process block, the "until condition" is tested and if found to be false, execution loops back to repeat the process block before testing the condition again.*

An Example

Now using the basic structures, we can construct a program of any size and complexity by combining and nesting in any manner as long as some fundamental rules are adhered to:

- The program as a process should have only one input path and one output path.
- Structures within the program can be nested but each structure must be totally contained within the structure in which it is being nested (this will be illustrated later).
- There should be no branching unless it is part of a structure (for example, the GOTOs required in languages like BASIC).
- Refrain from attempting to optimize the program by violating the above rules. There is a right time for this later.

Before we look at an example of structuring a program, let's first look at how nesting of basic structures works. Figure 7 shows a flowchart of a program which, overall, could be represented by a single *if-then-else*. But when it is looked at in more detail, the *else* leg contains another *if-then-else* as part of the instruction sequence there; the *else* leg of that structure contains yet another *if-then-else*. The heavy outlines show that each of the nested structures are totally enclosed by their parent structures; there is no overlap. A BASIC-like program to perform the function shown in figure 7 appears as listing 1. Again, I use outlines to illustrate that each structure is embedded in its entirety within another higher level structure. Notice that I have used indentation of lines to increase the readability of the program. Each separate structure should be at a different level of indentation than its parent.

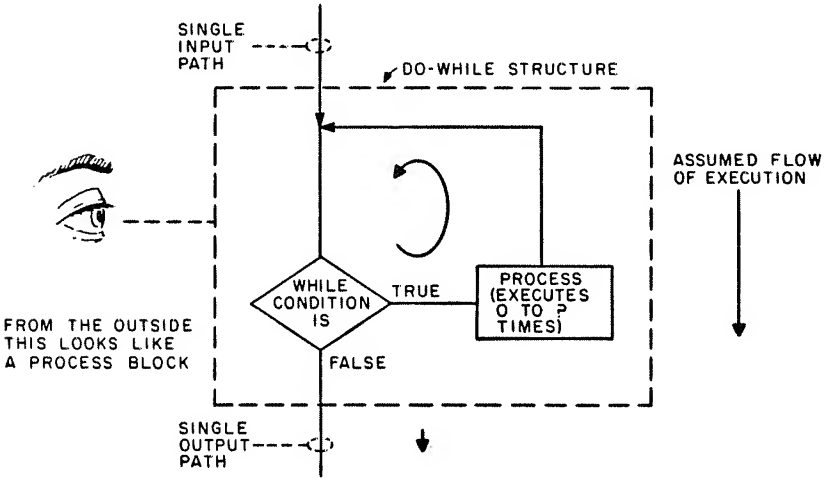
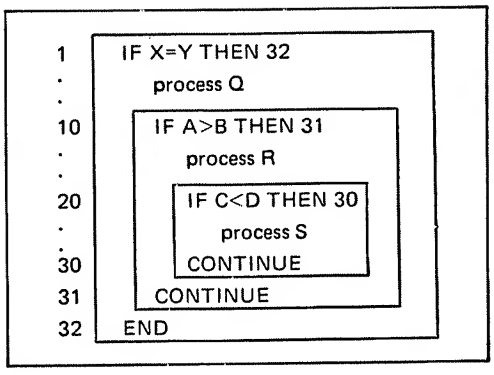


Figure 6: The do-while *structure is a looping form whose purpose is to execute a given process block only if the "while condition" is true. Thus it can execute the process block zero times if the condition is false initially, or an arbitrary number of times so long as the condition remains true during repeated execution of the process block.*



Listing 1: A BASIC-like program equivalent for the flowchart of figure 7. The lines in the picture emphasize the structured programming formalism.

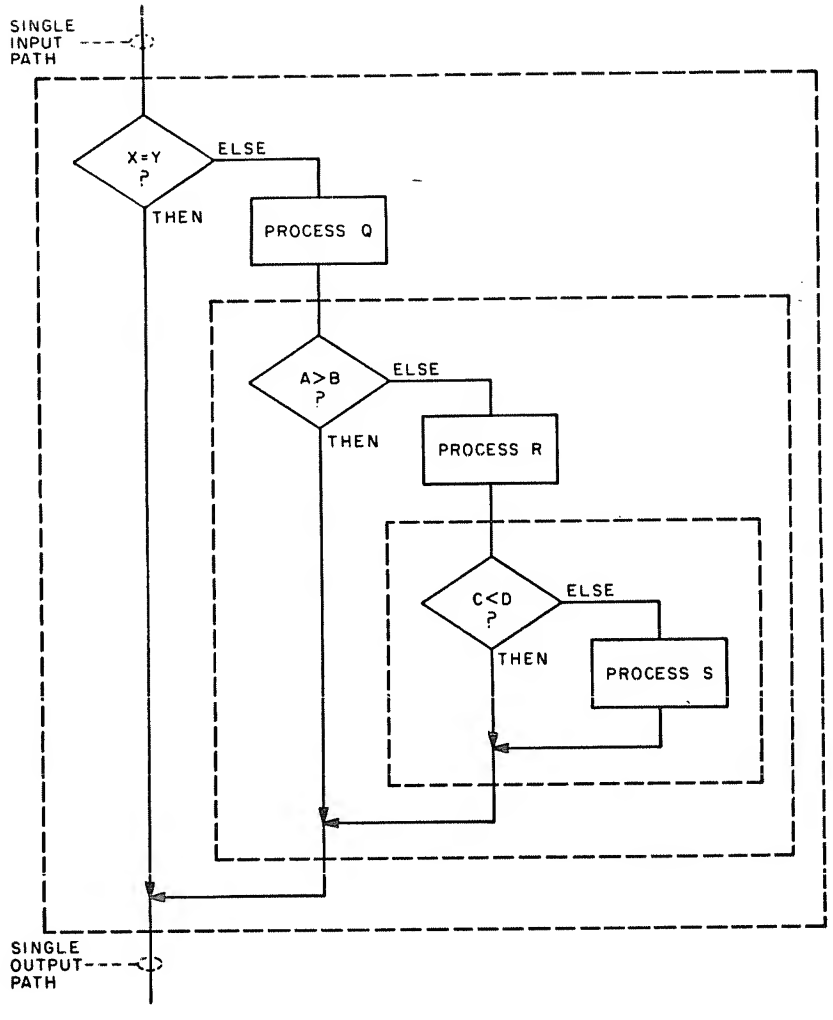
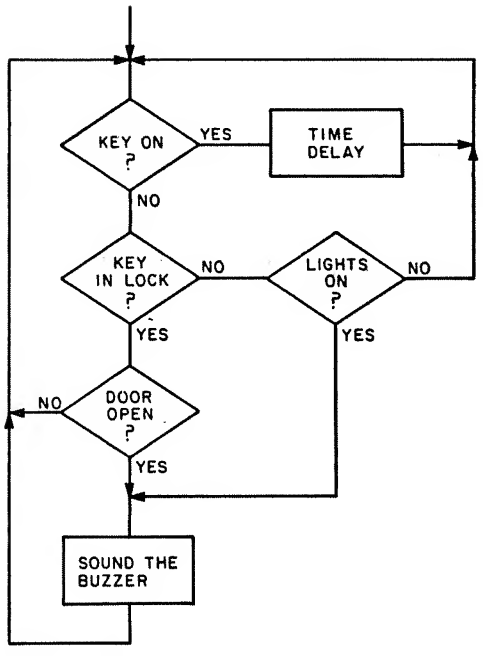


Figure 7: The various types of structures can be nested by noting that any place where a process block is indicated, a more complex structure can be used since it, too, only has one input path and one output path of execution. Thus, for example, this flowchart shows nesting of a process Q block and an if-then-else *structure as the else part of the* if-then-else *structure with condition* X=Y?. *This second* if-then-else *in turn has a third* if-then-else *as part of its else part. The outlines show the nesting of one structure within another.*

Figure 8: An unstructured flowchart performs an endless process as might be implemented in an automobile interlock. This is a complete and viable solution of the problem, but it involves numerous branching operations performed in an uncontrolled (GOTO) fashion.



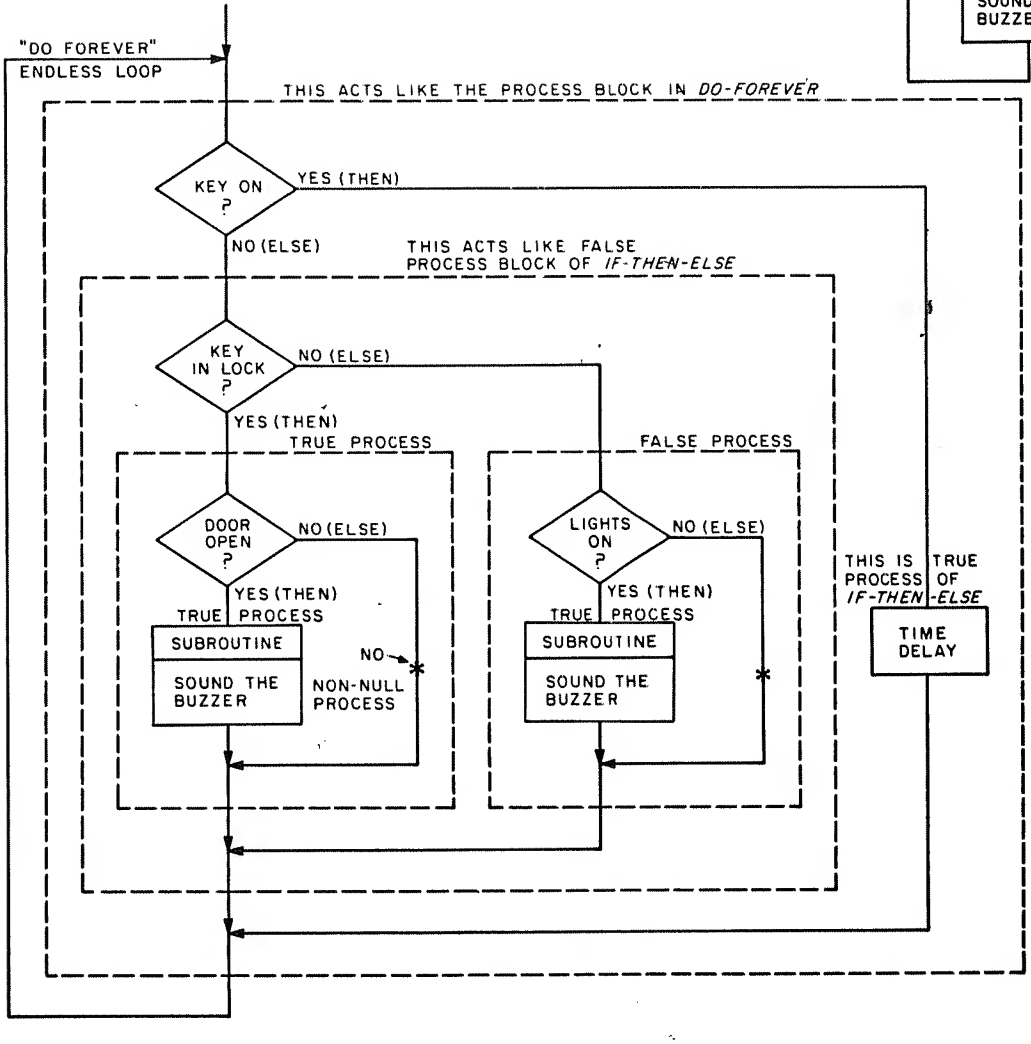


Figure 9: Taking the algorithm of figure 8 and casting it into a standardized, structured programming form eliminates all GOTO operations in languages with a complete if-then-else *structure, and in languages like BASIC, reduces use of GOTO operations to standardized structures. In this flowchart, we've positioned all the blocks to emphasize the nesting of structure. One of the primary reasons for the emphasis on structured programming is one of communications of ideas to other programmers (or the originating programmer at a later date). The claim is made that a flowchart like this one, and its equivalent representation in listing 2, provide a standardized way of communicating algorithms which makes the listing or chart easier to understand and read.*

Let's look now at an example of a simple program and show how a structured version might differ from an unstructured version.

The program is one which might be part of a future automobile computer control system using a microprocessor. Its purpose is to trigger a buzzer if the ignition key is left in the lock when the left front door is opened, or if the headlights are left on when the key is not in the lock. A delay is performed before conditions are checked again.

The flowchart in figure 8 shows how we might have drawn it without attempting to apply any of the principles of structured programming. Now, look at figure 9 which shows the structured version. Both forms of the program do the same function, but the structured form is clearly more straightforward and easier to write code from.

Basically, a number of things happened to the flowchart when it was structured. First, all the branches (or GOTOs) became forward branches except those in loop structures. This allows for reading the chart from top to bottom in an orderly way. Secondly, each decision block and process block has been put into a proper structure and nested totally within its parent structure. Thirdly, every structure regardless of its place in the overall program has only one input and one output.

One thing has happened that might appear to be a little strange to you. The sequence structure which performs the buzzer function appears twice now, where it only appeared once before. This is necessary in order to keep the structure clean. Remember, you cannot simply branch into the other buzzer block because those two structures would then overlap. The inefficiency implied by the double appearance of that block might bother you, but it will probably turn out that the block will be written as a subroutine and the only inefficiency will be an extra call instruction.

```
 1  LET X=0
 2  IF KEY ≠ ON THEN 7
 3      LET X=X+1
 4      IF X=5000 THEN 6
 5          GOTO 3
 6      GOTO 13
 7  IF KEY=INLOCK THEN 11
 8      IF LIGHT ≠ ON THEN 10
 9          GOSUB BUZZ
10      GOTO 13
11  IF DOOR ≠ OPEN THEN 13
12      GOSUB BUZZ
13  CONTINUE
14  GOTO 1
```

Listing 2: A BASIC-like application program for activating a buzzer of an automobile given several conditions. A subroutine BUZZ is indicated (by a call with the keyword GOSUB) to actually sound a noise during the loop. In this BASIC-like representation, several liberties with syntax have been taken.

Listing 2 is a BASIC-like program for the structured flowchart. (Here "BASIC-like" means using the syntax of BASIC but allowing variable names to be many characters in length for purposes of illustrating their meaning.) I have not attempted to make the program complete and have taken some liberties in order to illustrate my points.

A few words of explanation are in order. First, the instructions at lines 3, 4 and 5 represent a *do-until* structure which is used to implement a delay by simply incrementing a counter (X) until it reaches a large value. The name BUZZ represents the line number of a subroutine (not shown) which activates an electronic buzzer in the car's dash.

Now is the time to go back and look at the program to make it more efficient in its operation or in the amount of memory required. This should be done only if it is absolutely necessary. If it is necessary, try to maintain the structuring to the extent that it doesn't destroy the clarity of the program or increase its complexity. In our example program, notice that there are three CONTINUE instructions at lines 13, 14 and 15 leading to a GOTO at line 16. The speed of the routine can be improved and the memory requirements can be reduced by eliminating the CONTINUEs and changing any instruction which references any of them to go to line 16. Alternatively, you could change each of those references to go directly to line 1 although you would be seriously interfering with the intent of structuring.

In conclusion, I invite you to try the techniques described in this article when you write your next program. If you have done it any other way before, it takes a little getting used to, but I think you will ultimately agree that it has a lot to offer. Hopefully, you will see the benefits in the form of less time spent getting your program designed, written and debugged. In short, I believe that it can help make programming even more enjoyable.■

Applied Structured Programming

. . .and How to Use It: Part 1

Gregg Williams

Regardless of whether you're a newcomer to computers or a devoted computer enthusiast who occasionally manages to dream in hexadecimal, one thing is true: there's always room for improvement in your programming. Unless you are exceptionally orderly, you probably dive right into flowcharting or coding a problem, erase and do a lot of filling in when you remember something you hadn't thought of earlier, wind up with a final program where insertions choke your original code like weeds in a garden, and spend too much time correcting mistakes you think you shouldn't have made. And when you finish the program that (you think) is finally running, you complain to yourself, "There has got to be a better way!"

There is, and it's called structured programming. No, it isn't a language, and it isn't just a way to write a program. It's a philosophy of design that pays close attention to how a program comes to be written and tries to suggest ways to do each step more efficiently.

Structured programming evolved less than ten years ago, when computer programmers finally faced programs too big to understand, when the cost of writing and debugging software began to exceed the expense of using extra hardware and computer time. The school of structured programming evolved after E W Dijkstra voiced several thought provoking opinions, one of which was that many of our programming problems are caused by (over)use of the unrestricted GOTO statement, which is present in every high level language from COBOL to FORTRAN. And many people nodded their heads enthusiastically, for who hasn't traced the bug in a program to an unexpected juxtaposition of GOTOs?

Structured programming seems to help the habitual problems of even the most conscientious programmer, problems like how to write and debug a large program, how to fix (or better, to keep from happening in the first place) bugs in programs that crash after working correctly for months or years, or how to add to an already working program without causing unexpected side effects. But how do you do structured programming in a language that permits unlimited GOTOs?

Specifically, how do you do structured programming in BASIC, which is the universal language for the microcomputer user? Simple – you use GOTOs to implement the three basic structured programming structures that can theoretically represent any problem – sequence, *do. . .while* and *if. . .then. . .else* – and use GOTOs for nothing else.

What's the catch? You have to make yourself do it. The trade-off is simple: some discipline, a bit more planning in the early stages of designing a program, maybe a few extra lines of code in exchange for less total time spent in programming and getting a new program to work, less time spent debugging, and less chance of unexpected "blowups" happening later. It does seem to take more time, but that's because your lazy brain is protesting the exercise of little gray cells in thinking out a program and applying discipline; but the total time can be less, and the total frustration less.

I've been doing structured programming for some time as I write this (I got into it largely due to the complexity of the programming job I have at work), and now I wonder why anyone gets let out of programming classes without learning structured programming as the Gospel according to Dijkstra. Still, I've found several places where by-the-book structured programming is a bit awkward; and since I do have GOTOs to work with in BASIC, I found it hard to justify not using them when they can be used to simplify a program while still retaining the properties of "straight" structured programs.

This article, then, will deal with using structured programming in BASIC (with

Inversion Line	
=	≠
>	≤
≥	<
and	*or*

(a)

Conditional Expression	Its Inverse
X3 > 5	X3 ≤ 5
X = 5 *or* Y ≤ 0	X ≠ 5 and Y > 0
R ≥ (S + 3)	R < (S + 3)
K2 > K3 *and* K2 > K4	K2 ≤ K3 *or* K2 ≤ K4

(b)

Figure 1: Generating the inverse of a conditional expression. When converting structured pseudocode to BASIC code, the logical inverse of a conditional expression must often be inserted in an IF statement. At (a) is a table where each line contains two symbols, each of which is the inverse of the other. The rule for creating the inverse of a given expression is to replace every occurrence of the eight symbols in (a) with its inverse; (b) shows some examples of this.

emphasis on the word *using*) and with some extra programming structures I have found useful.

Some Preliminaries

Before I begin, I need to get two points out of the way. The first has to do with a property of the three basic control structures that must be duplicated by any proposed control structure for the latter to be suitable for a structured program. (I'm pointing this out to justify my additions and modifications to the three basic control structures.) In a word, each of the three structures in strict structured programming has the property called "one-in, one-out." This means that every time control of the program passes through this block (I will be calling the code between the boundaries of a control structure a "block"), it always starts with the same (first) statement and always exits through the same (last) statement – in other words, only one way in, only one way out. This allows a program to be constructed like a series of beads on a string, each of which can be examined and changed without inadvertently changing any of the others. (This is another property of structured programming control structures, the *functional independence* of each module. Given the same input, a module should perform the same operations regardless of what has happened in previous blocks.)

The second point is simply one of definition. In structured programming, we have situations where a block of code is done when a certain relationship holds true; if it is false, we do not do that block of code. These relationships, called conditional or relational expressions, are true or false depending on the current values of variables contained in the expression; examples are X1 = Y1, B>3, D+K1 ≠ K2. In structured programming, we do a block of code when a certain condition is met; to express this in BASIC, we must use the IF statement to branch around the same code when the condition is not met, that is, when the logical opposite or inverse of the same expression is true.

Several examples will help you here. When is X>5 false (for what values of X)? When is K1 ≠ 3 false? When is C≥100 false? The answers are respectively when X≤5, K1 = 3, and C<100. Why? Because the opposite of "greater than" (as in X>5) is "less than or equal to"; the opposite of "not equal to" (as in K1 ≠ 3) is "equal to"; and the opposite of "greater than or equal to" (as in C≥100) is "less than." And the converse is true as well, that is, the opposite of "less than or equal to" is "greater than," and so on. This also works with interchanging the logical connectives AND and OR (for example, the opposite of "G>5 AND A1 = 0" is "G≤5 OR A1 ≠ 0"). The justification for this line of reasoning can be found in any book on elementary logic (see DeMorgan's Law or DeMorgan's Theorem as it is variously known).

Because of all this, a simple table (see figure 1) allows us to find the inverse of a conditional expression. The rule to apply is: for a given expression, replace every occurrence of the symbols <, >, ≤, ≥, =, ≠, AND and OR, with the other symbol in the same row; the new expression is now the inverse of the first conditional expression.

Putting It in BASIC

Once we have the three basic control structures of sequence, *if...then...else* and *do...while*, we can look back to the moment before their invention and say, yes, because we are time bound creatures tied to the idea of serial or time ordered

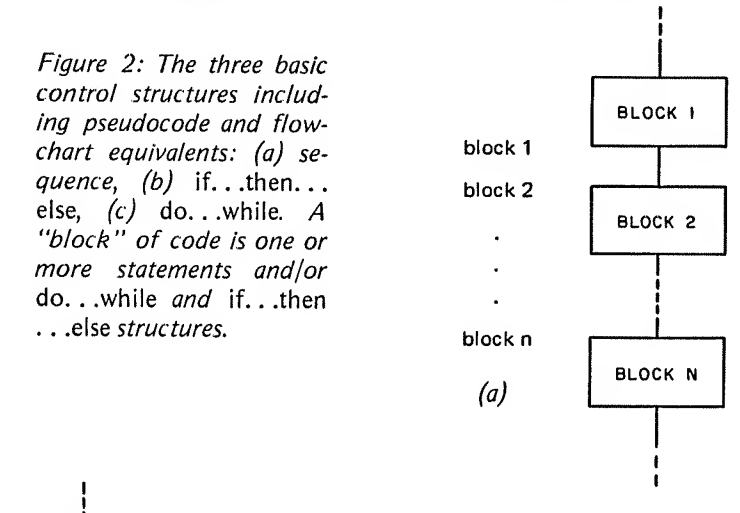
cause and effect, how else could we do anything? One either does tasks in sequence, or does a task until it no longer needs doing, or does one thing if something is true and another thing if it is not. How else can you decide on how to do a thing? (Unfortunately, when people stopped doing things by hand and programmed a computer to do them "for them," this intuitive causality was sometimes left behind. It's fitting that we returned to this intuitive causality only when the problem of writing computer programs was itself attacked as a problem.)

The three basic control structures written in convenient pseudocode are listed with their flowchart equivalents in figure 2. Note that, in an *if...then...else* sequence, in no way can both blocks 1 and 2 be done during the same pass, and that the decision whether to do a block 1 through n times in the *do...while* is made at the beginning of the block so that it is possible for a *do...while* block to do the enclosed blocks zero times.

Given these three control structures, every problem must be broken into varying levels of subproblems, each of which can ultimately be expressed as a combination of straight sequence, *if...then...else* sequences and *do...while* loops. How would you initially break these problems down using the above control structures?

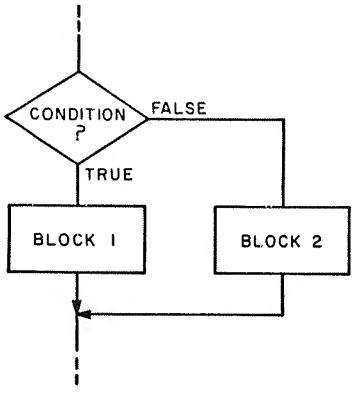
1. Given a number N, print the number, its reciprocal, and −1 times the number;
2. Average five test scores A, B, C, D, and E, to an average of V, including a 5 point curve if the initial average is below 70 points;
3. Print out the reciprocals of the numbers 1, 2, 3, . . . while the reciprocals are greater than 0.005.

Figure 2: The three basic control structures including pseudocode and flowchart equivalents: (a) sequence, (b) if...then... else, (c) do...while. A "block" of code is one or more statements and/or do...while *and* if...then ...else *structures.*



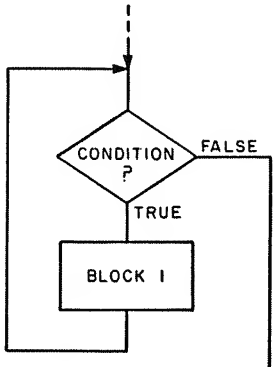
block 1
block 2
.
.
.
block n

(a)



if (condition)
  then
    block 1
  else
    block 2
  endif

(b)



do while (condition)
  block 1
endwhile

(c)

Figure 3: Three problems with their solutions in structured pseudocode (see text).

```
print N
print 1/N
print -N
```

Given a number N, print the number, its reciprocal, and −1 times the number;

(a)

```
V=(A+B+C+D+E)/5
if V < 70
  then V=V+5
  endif
```

Average five test scores A, B, C, D, and E, to an average of V, including a 5 point curve if the initial average is below 70 points;

(b)

```
N = 1
do while (1/N) > 0.005
  print 1/N
  N = N + 1
  endwhile
```

Print out the reciprocals of the numbers 1, 2, 3, . . . while the reciprocals are greater than 0.005.

(c)

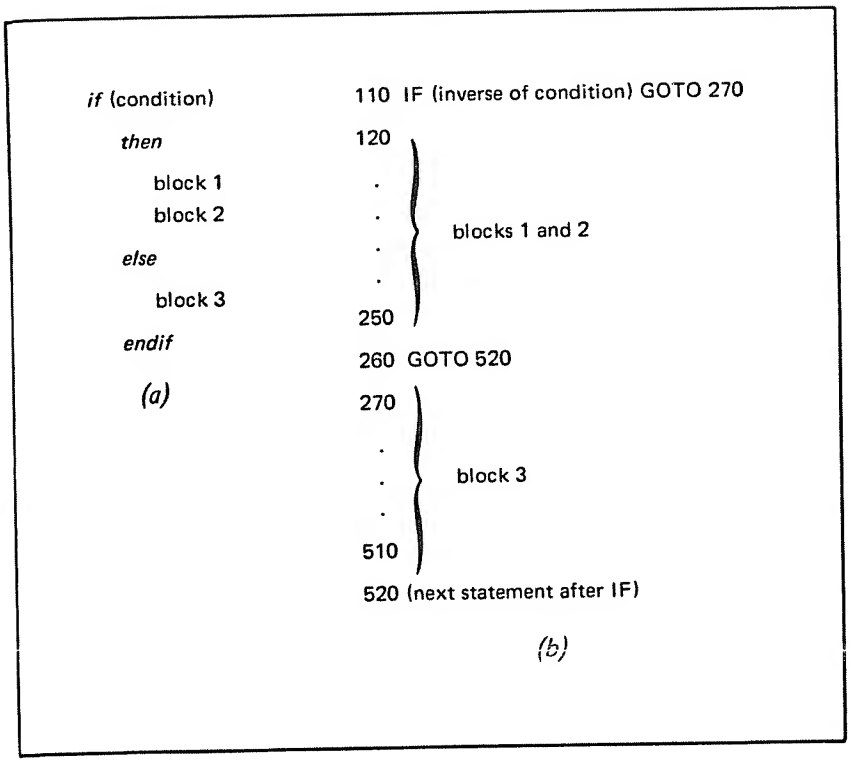


Figure 5: The if. . .then. . .else *structure: (a) pseudocode, (b) BASIC equivalent. The first statement (here line 110) is always an IF statement branching on the inverse of the condition given in the pseudocode; the branch is made to line 270, two lines past the last line performed by the "then" branch (line 250). The next statements are the code represented by the "then" branch (here lines 120 to 250), followed by a GOTO to the first statement after the "else" code (here line 260, branching to line 520). After the GOTO is the code for the "else" branch of the* if. . .then. . .else *statement (here lines 270 to 510). In actual practice of course appropriate line numbers would be used in the BASIC program.*

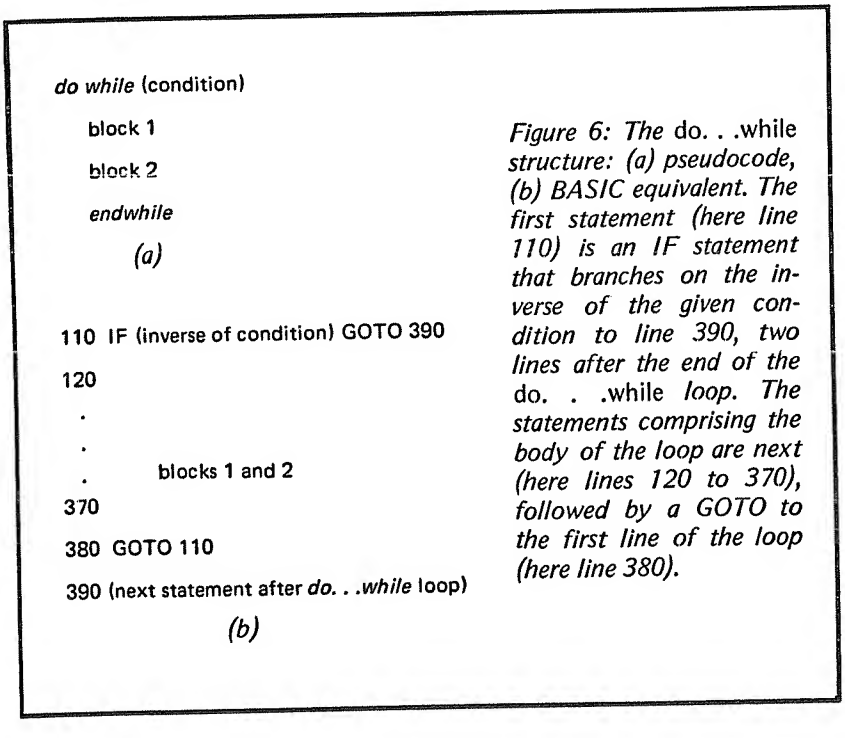


Figure 6: The do. . .while *structure: (a) pseudocode, (b) BASIC equivalent. The first statement (here line 110) is an IF statement that branches on the inverse of the given condition to line 390, two lines after the end of the* do. . .while *loop. The statements comprising the body of the loop are next (here lines 120 to 370), followed by a GOTO to the first line of the loop (here line 380).*

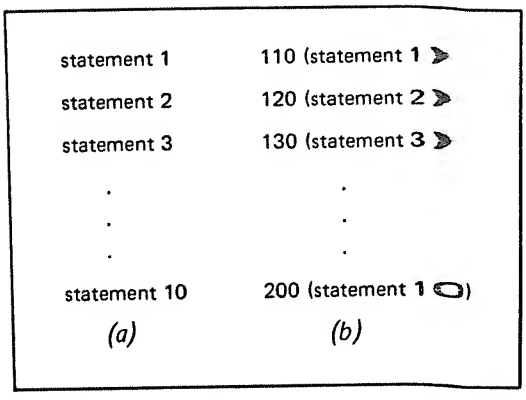


Figure 4: The "sequence" structure. The translation from pseudocode (a) to BASIC code (b) is simply one of coding several lines in ascending sequence.

(The answers are in figure 3.)

Once the idea of solving problems in control structure forms becomes natural, coding the problem in BASIC is no more than a straightforward translation (see figures 4, 5 and 6). Notice as mentioned before, that it is the inverse of the condition in the *do. . . while* and the *if. . .then. . .else* that appears in the BASIC code; this is because BASIC uses conditions for jumping instead of for not jumping. Except for that, coding structured BASIC is no more than a matter of practice.

After enough time for structured programming to become second nature to me, I found that certain applications of strict structured programming resulted in programs that were overly bulky or inelegant. Take the example of a program that sums up user entered numbers until a zero is encountered. The structured pseudocode and BASIC equivalent are shown in figures 7a and 7b. But notice that flag F1 exists only to signal that the *do. . .while* loop should be terminated immediately, a situation fully determined by whether or not the last input N is zero. The test of N in statement 150 is the second thing done in the loop that goes from 130 to 190; if control could transfer at the end of the loop to 140, which drops into the test at 150, we could throw away F1 and the *do. . .while* loop at 130, as in 7b, for a savings of one variable and several lines! This happens a lot in programs that interact with the user, so I thought, what if I devise a structure called "read X and do while (condition of X)?" It is still one-in, one-out; it's easy to understand; and it's barely different from a plain *do. . .while*. So I used it and began looking for other opportunities to add constructs

Figure 7: Improving "strict" structured programming: (a) is the pseudocode for a problem to add user inputs until a zero is encountered, using only sequence, if. . .then . . .else *and* do. . . while. *(b) is the pseudocode of (a) translated into BASIC. (c) is an equivalent BASIC solution that saves three lines of code and one variable by slightly bending the form of a structured programming* do. . .while *loop.*

```
sum = 0
flag = 1
do while flag=1
   input num
   if num≠0
      then
         sum=sum+num
      else
         flag=0
      endif
   endwhile
(next statement)
```

(a)

```
110 S = 0
120 F1 = 1
130 IF F1≠1 GOTO 200
140 INPUT N
150 IF N=0 GOTO 180
160 S = S + N
170 GOTO 130
180 F1 = 0
190 GOTO 130
200 (next statement)
```

(b)

```
110 S = 0
140 INPUT N
150 IF N=0 GOTO 200
160 S = S + N
170 GOTO 140
200 (next statement)
```

(c)

to structured programming as originally conceived.

Do. . .until

The structure closest to the three basic control structures is the *do. . .until* loop, illustrated in figure 8. The main difference between it and the *do. . .while* loop is that, in the *do. . .until* loop, the expression to be evaluated is the last statement in the loop instead of the first; this insures that the body of the loop is done at least once.

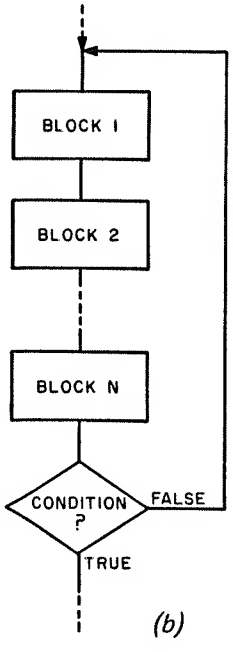
A *do. . .until* loop is written in BASIC by writing the statements in the body of the loop, then adding an IF statement that branches to the first statement of the loop if the condition is not met (the inverse of the original conditional expression is used here).

Consider our earlier problem of adding a number of user inputs until a zero is encountered. The solution to this, using the *do. . .until* structure, is given in figure 9. Notice in the pseudocode that I've put the conditional relation on the last line of the

Figure 8: The do. . .until *loop: (a) pseudocode, (b) flowchart, (c) BASIC equivalent. The first statements (here lines 110 to 380) are the code for blocks 1 thru n. The next and last statement (here line 390) is an IF statement that branches on the inverse of the condition listed in the pseudocode to the first statement of the* do. . .until *loop (here line 110).*

```
do until test
   block 1
   block 2
      .
      .
      .
   block n
   endif (condition)
```

(a)



(b)

```
110
120
 .
 .
 .
380
390 IF (inverse of condition) GOTO 110
400 (next statement)
```

blocks 1--n

(c)

Figure 9: Illustration of do. . .until *loop: (a) pseudocode, (b) BASIC equivalent. The problem illustrated is to create code that will sum all user inputs until a zero is encountered, then print the sum.*

```
S = 0                              100 S = 0
do until test of N
    input N                        110 INPUT N
    S = S + N                      120 S = S + N
    endloop if N = 0               130 IF N≠0 GOTO 110
print S                            140 PRINT 'SUM IS'; S
```

(a) *(b)*

loop to remind the user that the test is made at the end, not the beginning, of the loop. You may prefer to write the conditional expression on the same line as the words *do until* (in this case, line 2 of the pseudocode as *do until N = 0*), but you will have to remember that the test is delayed until after the last line of the loop.

Looking at the BASIC code, there is a one-to-one correspondence between the pseudocode and BASIC code except that line 2, the *do until* line, has no equivalent, and line 5, the *endloop* line, translates into an IF statement that completes the loop only if N ≠ 0 (that is, only if the inverse of the conditional statement is true). (Looking back at figure 7c, we see that we have improved on our *read N and do until N = 0* loop by one statement, mainly because a *do until* will always have one less BASIC statement than its corresponding *do while*.)

Case

The *case* statement is used when the value of a variable determines which of N mutually exclusive blocks of code is to be exe-

```
case on N
    if N=1, block 1
    if N=2, block 2
    if N=3, block 3
    endcase
```

(a)

```
110 GOTO 120, 200, 250 ON N
120
 .
 .   block 1
 .
180
190 GOTO 320
200
 .
 .   block 2
 .
230
240 GOTO 320
250
 .
 .   block 3
 .
310
320 (next statement)
```

(b)

```
110 IF N≠1 GOTO 190
120
 .
 .   block 1
 .
180
190 IF N≠2 GOTO 240
200
 .
 .   block 2
 .
230
240 IF N≠3 GOTO 320
250
 .
 .   block 3
 .
310
320 (next statement)
```

(c)

Figure 10: The case *statement: (a) an example in pseudocode, (b) the example in BASIC using a computed GOTO, (c) the example using a series of IF statements. The computed GOTO (b) is used when the values that N takes can be "boiled down" to the integers 1, 2, 3, . . . (for example, if N took the values 10, 15, 20, we would GOTO on (N-5)/5). A series of IF statements (c) would be used when the values of N are irregular and (b) cannot be used.*

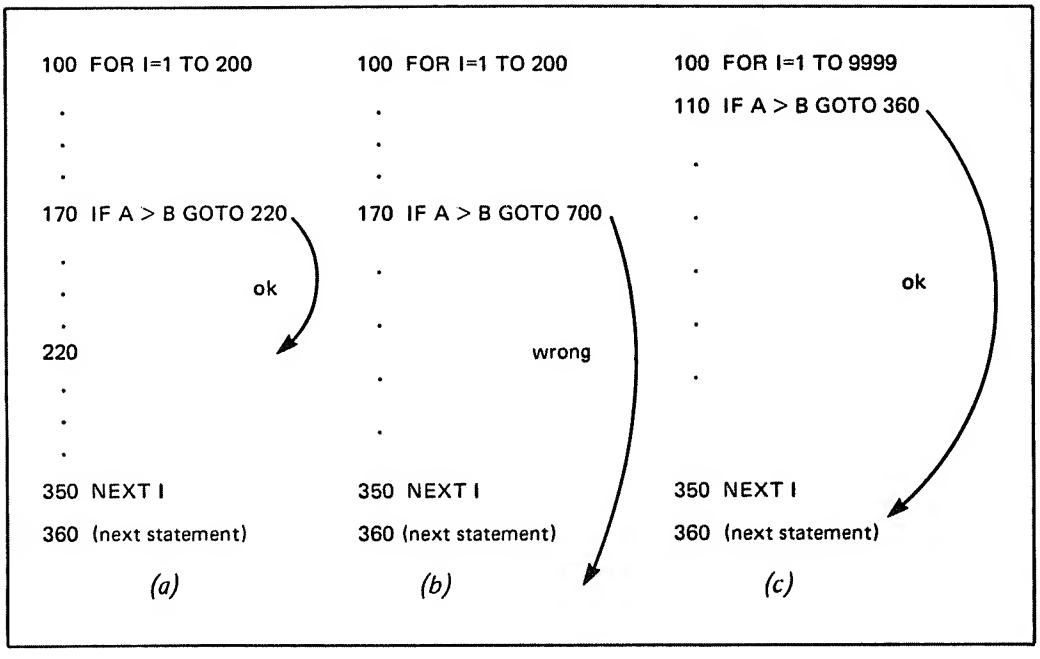
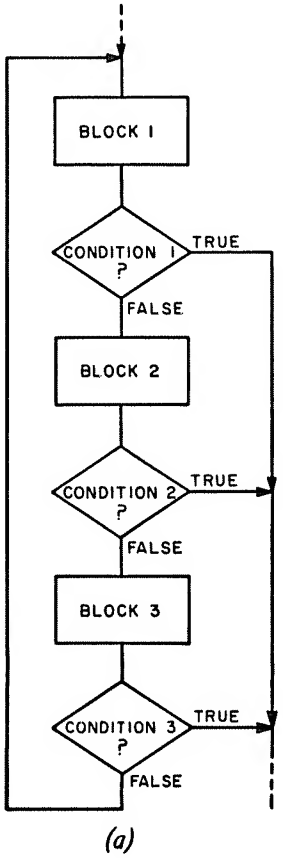


Figure 11: Uses of FOR. . .NEXT loops in structured programming. A FOR. . .NEXT loop is okay if no statement within the loop ever transfers control outside the main loop; see (a). The situation in (b) is definitely not structured; there is no way to guarantee that line 360 (and subsequent lines) will be done when the FOR. . .NEXT loop is completed. A do. . .while *loop may be fashioned as in (c), which is equivalent to* do while $A \leqslant B$. *Note that the index of the loop, I, is simply "marking time" but as such cannot be used for another purpose within the loop.*



(a)

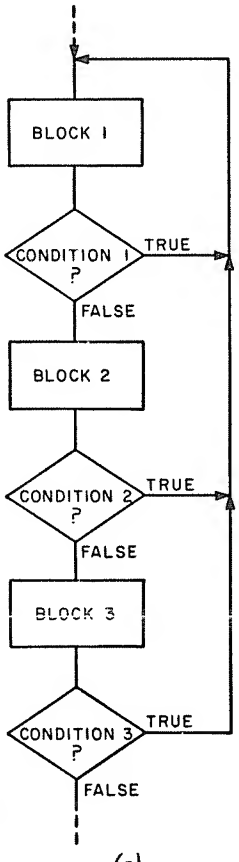
beginloop
block 1
exitif (condition 1)
block 2
exitif (condition 2)
block 3
endloop

(b)

```
110
 .
 .   block 1
 .
150
160 IF (condition 1) GOTO 470
170
 .
 .   block 2
 .
310
320 IF (condition 2) GOTO 470
330
 .
 .   block 3
440
450 IF (condition 3) GOTO 470
460 GOTO 110
470 (next statement)
```

(c)

Figure 12: The beginloop . . .exitif. . .endloop *structure: (a) flowchart, (b) pseudocode, (c) BASIC equivalent. Notice the IF statements (here at 160, 320 and 450) all branching to the first line of the next structure (here line 470, two lines after the end of block 3). The second line after the end of block 3 (here line 460) is a GOTO that jumps to the first statement of block 1.*



(a)

```
beginblock                      110
    block 1                      .
                                 .   block 1
                                 .
loopif (condition 1)            150
    block 2                     160 IF (condition 1) GOTO 110
loopif (condition 2)            170
    block 3                      .
                                 .   block 2
                                 .
loopif (condition 3)            310
endblock                        320 IF (condition 2) GOTO 110
                                330
                                 .
                                 .   block 3
                                 .
                                450
                                460 IF (condition 3) GOTO 110
                                470 (next statement)
```

(b) (c)

Figure 13: The begin-block . . . loopif . . . exit-block *structure: (a) flowchart, (b) pseudocode, (c) BASIC equivalent. Notice that the IF statements (here at 160, 320 and 460) go to the beginning of block 1 and that they branch on the condition itself, not the inverse.*

cuted (with control passing to the next statement after the chosen block is performed); from this, you can see that the *if. . .then. . .else* structure is a special *case* statement with N = 2.

A *case* statement is implemented in BASIC by either sequential IF statements or, if the variable can be "boiled down" to an integer ranging from 1 to N, a computed GOTO statement. Remember that, since control eventually passes to the first statement after the *case* statement, no block within the *case* statement may contain a GOTO statement except as the last statement within a block branching to the first statement after the *case* statement; to do otherwise would damage the structure's property of one-in, one-out.

An example of a *case* pseudocode statement and two BASIC equivalents is given in figure 10. Note that, when using a computed GOTO, each block of code must end with GOTO nnn, where nnn is the next line after the *case* statement. In figure 10c, IF statements are used to branch around the blocks of code if the variable N does not have the appropriate value for that block.

Subroutines, User Defined Functions, and FOR. . .NEXT Loops

One of the most important features of a structured program is that it is composed of one-in, one-out blocks that are not jumped into or exited from except at the beginning or the end of the block. Therefore, as far as I am concerned, there is no reason from the structured programming point of view why I can't use both subroutines and user defined functions (using the DEF statement) in my structured programs; they are both one-in, one-out constructs of BASIC and save repeating identical code.

Using the FOR. . .NEXT loop is a different matter. Unlike the subroutine or the user defined function, control can be transferred from anywhere inside the loop to anywhere outside the loop; in this case, a FOR. . .NEXT loop by itself is unsuitable for a structured program and should be replaced by either a *do. . .until* or a *do. . . while* loop (if used properly, a FOR. . . NEXT loop can be used to implement either of these; see figure 11). But a FOR. . .NEXT loop's most valid use is

simply as a shorthand for a block of code to be repeated identically a given number of times. Used this way, the loop keeps the one-in, one-out feature necessary to all structured programming control structures.

Beginloop. . .exitif. . .endloop

The *beginloop. . .exitif. . .endloop* structure is described in several books detailing advanced structured programming techniques, and while it does not have the gut level intuitive appeal the basic three do, it keeps popping up in programs I write, so it must be fairly useful.

The flowchart for the *beginloop. . . exitif. . .endloop* structure is in figure 12a. It is basically a loop with several exit points (*do. . .while* and *do. . .until* can be seen as specific cases of this general form). It has one entrance and one exit (several exit points, but the transfer of control is always to the next statement after the loop structure), and an example in pseudocode and BASIC is shown in figures 12b and 12c. Note that here the conditional expression and not its opposite is translated from pseudocode into BASIC (see lines 160, 320 and 450, figure 12c).

Other Structures

Even with all the above structures, I keep finding situations that can't be fitted into any of them. So when several situations came up repeatedly, I modified existing structures to fit them and still be of the most general use. But, for the structures that remain, the emphasis is more on convenience than on utility.

A very useful variation of the *beginloop. . .exitif. . .endloop* structure is one that loops (instead of exits) when certain conditions occur. I call this a *beginblock. . . loopif. . .endblock* structure (see figure 13); it is very useful for performing a certain operation until all of a series of conditions are met. An example of this is the code in figure 14 that requests from the user an integer input between 1 and 10; notice that we *loopif* the input N is not between 1 and 10, and we also *loopif* N is not an integer.

I have created another pseudocode instruction called *read. . .until valid* for the specific purpose of reading and validating a user input, usually when the validation process is very simple. The BASIC code for the above problem is the same as in figure 14b (unless you want to add some error message statements), and the pseudocode is simply:

beginblock	
input N	110 INPUT N
loopif N≥10 or N<1	120 IF N>10 *or* N<1 GOTO 110
loopif N ≠ INT(N)	130 IF N ≠ INT(N) GOTO 110
endblock	
(next statement)	140 (next statement)
(a)	*(b)*

Figure 14: An example of beginblock. . .loopif. . .endblock: *(a) pseudocode, (b) BASIC equivalent. The problem illustrated is to get an input from the user that is both between 1 and 10 and an integer. Here the second block (between the two* loopifs*) is empty.*

---

read N until valid
invalid if N not between 1 and 10
invalid if N not integer

with the two invalid lines not necessarily written down. (Notice that for the last two structures, the conditional expressions used in the pseudocode are not inverted when transferred to the BASIC code.)

The Beginning of the End

Those are all the structures I've come up with. They may or may not be justified in your mind by the improvements they allow over "strict" structured programming; but each of them is (at worst) a shorthand that takes the programmer a step further from planning a program in instructions and a step closer to planning in well-defined subtasks. And because these subtasks are always a proper subset of any language that allows unlimited GOTOs, it is simple to write structured programs in BASIC (or in any other all-purpose language), using modules of code that are functionally independent and "one-in, one-out."

However, there are several other aspects of problem solving — problem definition, program design, debugging and testing, and program revision — that can benefit from the application of a methodical technique (and this becomes less of a luxury and more of a necessity as program size increases). In the second part of this article I'll use the problem of writing a game to play NIM to illustrate the use of structured programming in the entire problem solving process.■

Applied Structured Programming . . .and How to Use It: Part 2

Gregg Williams

In part 1 I covered the basic constructs of structured programming, several additional structures (see table 1), and how to program them in BASIC. Now I want to show my idea, at least, of good programming habits, as well as the application of structured programming techniques to the entire range of problem solving. As an example, I will show how I went about writing a program that plays the game of NIM.

NIM as a Computer Game

I picked NIM because it is simply analyzed, making it possible to concentrate on the writing of the program and not on the development of the computer's playing strategy.

Basic Structured Constructs

sequence
if. . .then. . .else
do. . .while

Added Structured Constructs

do. . .until
case
subroutines
for loops
beginloop. . .exitif. . .endloop
beginblock. . .loopif. . .endblock
read. . .and do while
read. . .until valid

Table 1: The constructs of applied structured programming, as explained in part 1. The "basic" constructs are universally recognized as being sufficient to implement any program; the "added" constructs are recommended by the author as extensions of the basic constructs that make structured programming more versatile and manageable.

The rules are as follows: the game starts with a pile of, say, 17 sticks. Players alternate turns, taking one, two or three sticks. The person taking the last stick loses.

It doesn't take much analysis to show that a player is in a "safe" position if there are 1, 5, 9, 13,. . .pieces after his or her move. No matter how an opponent moves, the player can take enough sticks (four minus opponent's move) to put the game back to a "safe" position. The player's opponent is hamstrung and will definitely lose.

The computer's strategy is given in table 2 and is based on what the pile of sticks looks like in terms of multiples of 4. The computer wants to leave the pile in the form (4n+1), a safe position for it. But the computer is in a bad situation if the pile looks like (4n+1) at the beginning of its turn (it also means the human player is in a "safe" position and will win if the correct moves are made). In this case, the moves of 1, 2 and 3 all leave the computer in an "unsafe" position; so, for this program, I decided to let the computer take 1 so as to prolong the game.

A Problem Solving Approach

Before I get into the hand waving that will enable you to see how I wrote this program, I'd like to give you an overview of how I think a program should be attacked à la structured programming.

Step 1: Define the program in terms of what it will and will not do. Don't laugh—who hasn't been coding a program only to remember, "Omigosh, I forgot to put in something to . . . ." Keeping last minute additions or afterthoughts to a minimum reduces the possibility of unexpected interaction between statements, often called bugs, glitches, blowups and so on.

Step 2: Flowchart "the big picture." A lot of this is intuitive, but it means break the program into the first subprograms that come to mind, and show where these come

in program flow. Unless a program is very simple, it is hard to go straight past this step into step 3; I have to literally see the program flow in flowchart form before I can begin thinking in terms of *if. . .then . . .else* and *do. . .while* and other control structures.

Step 3: Translate this into an overview with structured pseudocode. By now you have to have a loose idea of what the subprograms (let's call them modules) will do. You don't have to have module definition pinned down to the finest point, for simply having thought in terms of modules means you've already put more thought into this stage than most people do. Also, gluing the modules together with structured pseudocode gets you started toward a structured program. It won't be structured unless it starts structured.

Step 4: Program each module in pseudocode. Aha, here's where you find out what you've left out. Notice I said "program." I mean it: you should write out exactly how a module will be executed as if it were the program that goes into the computer. The pseudocode should be so detailed that translating it into BASIC (or any other language) is almost a mechanical chore. This step may cause you to go back to step 3, but that's okay, for it is probably faster to revise on paper than in the computer.

A note on modules: a key factor in the success of a structured program is the functional independence of modules. This means that a module should do a certain thing regardless of what the modules before it do, thus minimizing the possibility of unexpected module interaction. For example, if module A is designed to perform some computation on variables X and Y giving result Z, the only way module B, which calls module A, should be able to influence results is by changing the inputs X and Y prior to the call. The internal machinations of B should not affect A except through the identified input and output parameters of block A.

Step 5: Translate each module into BASIC code. Using the forms I outlined in part 1, going from pseudocode modules to BASIC modules is a mechanical translation process; the only thing you really need to think about is assigning and keeping track of variables and functions. I use a chart to do that; see figure 8 for an example.

Step 6: Test each module. You're on your own here, but you must do something to check out what a module is supposed to be doing in terms of function, input and output. This is a "bottom up" approach to programming (note, however, that the design is "top down"). Although "top down" programming has been praised for its ability to catch unexpected module interaction, I ask: how can it until those modules (mistakes and all) have themselves been written? (An incisive analysis of the design process is given by Knuth in *The Art of Computer Programming, Fundamental Algorithms,* volume 1, pages 187 to 189 in the second edition.)

Number of Sticks in Pile at Beginning of Computer's Turn	Number of Sticks Computer Takes	Resulting Position for Computer
4n	3	safe
4n+1	1	unsafe
4n+2	1	safe
4n+3	2	safe

Table 2: Computer's strategy for the NIM game. Either player is guaranteed a win (assuming that player makes no mistakes) if the pile of sticks is a number of the form 4n+1 at the end of his/her turn. Notice that when the play begins with 4n+1 sticks, the computer is forced to take one, two or three sticks and so leaves itself in an unsafe position at the end of its move.

Step 7: Test the program. Glue the modules together with the BASIC equivalent of the pseudocode from step 3. Start it running and hunt down bugs. Even if it works, keep hunting until you are tired of running the program. The brevity of this step (and the assurance that the program will not one day unexpectedly blow up) is your reward for the work done in the first six steps.

Step 8: If you add to the program, add structured code. I know it's hard to do. It's even hard for me to do, and I'm the one who's writing this article. But, unless the addition is extremely trivial, make sure that the code you add fits in, in a structured sense. Don't jeopardize functional independence. Do break down a module if necessary to rewrite it.

NIM: Initial Design (Steps 1 thru 3)

Now we're ready to work on the NIM playing program. After thinking about the possibilities, I decided on this rough working definition: This program will play a series of NIM games against a human opponent. It will use the residue of four algorithm for its strategy and will give the user the option of choosing who goes first and how many sticks are in the pile; the default will be 17 sticks and human goes first, an automatic win for the computer. The program will also check human inputs for validity.

My initial flowchart is in figure 1. Notice that there are four basic modules: *initialization, player-turn, computer-turn* and *evaluation.* If you want to go into more complex detail (and you will have to in a larger program), you can say that *initialization* basically sets the number of sticks in the pile and who goes first. *Player-turn* accepts a move, checks its validity, and subtracts the move from the pile. *Computer-turn* analyzes the pile, chooses a move, and subtracts the move from the pile. *Evaluation*

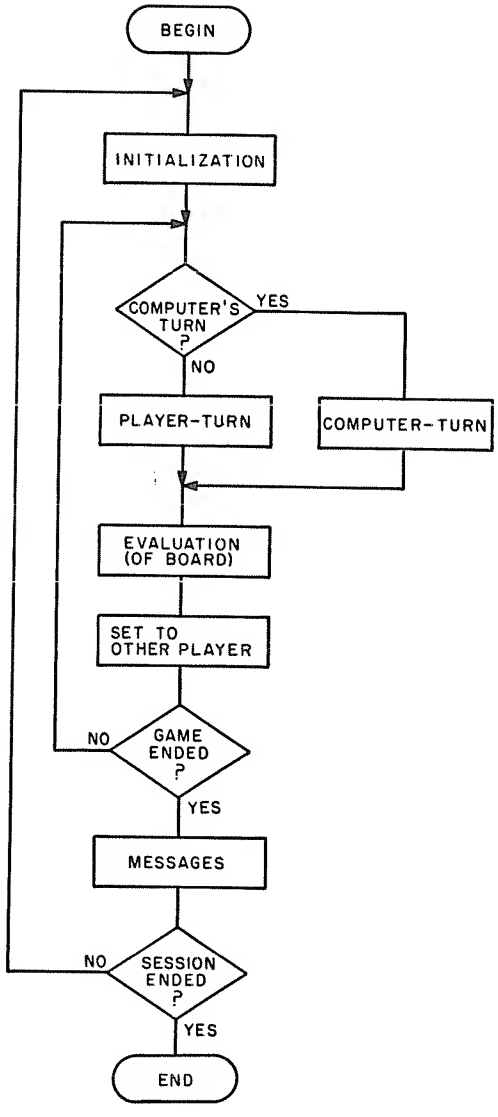


Figure 1: A high level flowchart for the NIM program. Most of the blocks represent some large chunk of the overall problem; such blocks are called "modules." Actions common to more than one block should be brought outside the blocks and shared. For example, the block labeled "evaluation" was part of both modules "player-turn" and "computer-turn" until it was seen that it could be brought outside and made a module of its own.

Figure 2: The structured pseudocode overview. This overview, equivalent to the flowchart of figure 1, is written in a pseudocomputer language and is the first step past the flowchart toward a completed BASIC program. Notice that the modules which will be filled in as details are later noted as names enclosed in parentheses, that preliminary variables are used and given descriptive names, and that the entire problem is outlined by this pseudoprogram.

```
0   NIM:
1       games-played = 0
2       do until test of endsession
3           do until test of gamewon
4               (initialize)
5               if computer's-turn = 0 then
6                   (player-turn)
7               else
8                   (computer-turn)
9               endif
10              (evaluate)
11              computer's-turn = 1 – computer's-turn
12              endloop if gamewon
13          print endgame messages and ask if user wants to play again
14          receive user response
15          if response = yes [ie: if endsession = 0]
16              games-played = games-played +1
17              endif
18          endloop if endsession
19      end-of-program
```

checks to see if the pile is down to 1 (in which case the winner is declared) and also makes several comments as endgame approaches (this is the only module whose function grew as the module was written).

The first real work is done with the creation of the structured pseudocode overview in figure 2. The process is fairly simple here because the program is well-defined as a flowchart; but with a flowchart that has constructs that are definitely not recognizable control structures, you have to twist the flowchart (maybe even rewrite it) until you can see it in terms of sequence, *if. . . then. . .else* and *do. . .while.*

Notice that at this step you begin to define flags. There will be a flag (which will have a value of 1 or 0, standing for true or false) representing the status of *endsession, gamewon* and *computer's-turn.* Notice also that variable names are descriptive enough for the reader to understand exactly what is happening; be sure to keep this first pseudocode overview readable.

NIM: Detailed Design (Step 4)

The bulk of thinking from here on out is in this step, the writing of each module of pseudocode. You will probably discover changes and additions you need to make; the advantage of doing so at this point is that it is easier to make corrections and revisions in pseudocode than it is to make them in BASIC. One reason for this is that, since pseudocode is not read by the computer, you do not have to spend any time making sure that it is syntactically correct (instead, you spend the same time making more changes); another is that pseudocode is easier to change because it is easier to read. Compare the pseudocode "if who-plays-first = computer, then. . ." with the BASIC statement "1600 IF P=0 GOTO 1660."

The pseudocode for my basic modules is in figures 3 thru 7. Although I made really only one draft of each module before I translated it to BASIC, the draft itself contains many erasures and insertions; for me, working in pencil is a must. You may find an operation that occurs several times within the different modules; if so, you'll want to make it either a module or a subroutine. In the case of the NIM program, I decided to make a module out of the part of code needed to print the current board position. I could have left it as part of the *evaluate* module, but to do so would have obscured the module's purpose with too much detail. As it stands, the only information given in the *evaluate* module is "Print current board

Figure 3: The initialize *module.*

Relative Line No	BASIC Line No	Pseudocode
0	—	initialize:
1	220	if games-played = 0
2	230	ask if user wants instructions
3	240	read user-answer until valid
4	260	if answer is 'yes'
5	280	print instructions
6	310	endif
7	310	endif
8	330	ask user if he wants to choose number sticks and who goes first
9	360	read user-choice until valid
10	380	if user-choice = default
11	390	number-of-sticks = 17
12	400	computer's-turn = 0
13		else
14	420	ask user how many sticks to begin with
15	440	read number-of-sticks and do while number-of-sticks <13
16	460	error message 'sorry. we have to have at least 13 sticks'
17	470	endwhile
18	490	ask user who goes first
19	500	read computer's-turn until valid
20	510	endif

The pseudocode in figures 3 to 7 represents the second level of breaking a problem into subproblems (the first level was from defined problem to the structured overview of figure 2). Notice that the trend is to write the lines so that they are easily understood rather than to make them look like formal computer code. The numbers in front of most of the lines represent the beginning line numbers of the equivalent statements in the BASIC program.

Figure 4: The player-turn *module.*

Relative Line No	BASIC Line No	Pseudocode
0	—	player-turn:
1	810	do until test of invalid-move
2	845	valid-move = 1
3	830	ask user for his move
4	840	input user-move
5	850	if user-move not between 1 and 3
6	855	print error message
7	860	valid-move = 0
8	860	endif
9	870	if user-move ⩾ number-of-sticks
10	875	print error message
11	880	valid-move = 0
12	880	endif
13	900	endloop if valid-move = 1
14	920	number-of-sticks = number-of-sticks — user-move

Figure 5: The computer-turn *module.*

Relative Line No	BASIC Line No	Pseudocode
0	—	computer-turn:
1	1120	remainder = number-of-sticks modulo 4
2	1140	case on remainder
3	1150	if remainder=0, then computer's-move=3
4	1170	if remainder=1, then computer's-move=1
5	1170	if remainder=2, then computer's-move=1
6	1190	if remainder=3, then computer's-move=2
7	1210	number-of-sticks = number-of-sticks — computer's-move
8	1230	print computer's-move to user

Relative Line No	BASIC Line No	Pseudocode
0	—	evaluate:
1	1415	(print-board)
2	1520	if number-of-sticks = 1
3	1540	if computer's-turn = 1
4	1550	print computer-loses message
5		else
6	1570	print user-loses message
7	1570	endif
8	1570	endif
9	1590	if number-of-sticks between 6 and 8
10	1600	if computer's-turn = 1
11	1620	if RND < 0.6
12	1630	print computer-resigns message
13	1640	number-of-sticks = 1
14	1640	endif
15		else
16	1660	if RND ≤ 0.3
17	1670	print you're-in-trouble message
18	1690	ask if user wants to resign
19	1700	input user-answer until valid
20	1720	if user-answer = 'yes'
21	1730	print user-resigns message
22	1740	number-of-sticks = 1
23		else
24	1760	print nasty answer
25	1760	endif
26	1760	endif
27	1760	endif
28	1760	endif

Figure 6: The evaluate *module. "RND" refers to a random number between zero and one; when either player resigns, number-of-sticks is set to 1 to signal end-of-game.*

position," which tells the reader exactly what is being done; if the reader wants more detailed knowledge, it is possible to refer to the *print-board* module.

The value of pseudocode can be seen in the fact that very little in the program needs to be explained. This is why high level languages which are closer to pseudocode make better programming languages than BASIC. Figures 3 and 4 are complicated only by *read. . .until valid* statements that check user responses (the checking of input data is usually a good idea unless your computer has real space problems). The *computer-turn* module, figure 5, implements the computer strategy of table 2.

In the *evaluate* module, figure 6, if both players play perfect games and the number of sticks is six, seven, or eight, the player who has just moved will definitely lose the game (the opponent can take one, two or three sticks, respectively, finishing with five sticks, a "safe" position for the opponent). The if statement beginning "If number-sticks between 6 and 8" (line 9 of figure 6) and ending with the last "endif" of the module takes care of this situation. If the computer is about to lose, it resigns six-tenths of the time; otherwise, it gives the human a chance to resign three-tenths of the time. (Notice, in figure 1, that the *evaluate* module comes before the variable *computer's-turn* is changed, so that, in the *evaluate* module, *computer's-turn=0* means that the human has just played and that the computer's turn is next.)

NIM: Translating to BASIC (Step 5)

Two aspects of the pseudocode-to-BASIC translation need attention: the translation itself (covered in Part 1), and the assignment of variable names and meanings. The translation, although it requires attention, is straightforward enough once you are used to it. Assigning BASIC line numbers corresponding to relative line numbers of pseudocode (given in figures 3 to 7) should make matters easier.

The assignment of variable names, however, is another matter. Each named variable or flag must be replaced with a letter or letter-plus-digit name. I think it is a good idea to keep track of what names have been used, what modules they are used in, and what they are used for. An example of the chart I usually make (here, for the NIM program) is in figure 8. I also try to decide whether or not I need to store a variable's value for use later in the program; if not, I can use the same variable later and save a few bytes of storage (when compared with creating and using a new variable).

When writing a module of BASIC code, I write the entire module, using a circled letter in colored pencil to link the space after a GOTO to the line number it belongs with. Then I number the entire module and replace the circled letters with the correct line numbers.

Comments are very important and, unless you are working with severe memory restrictions, there is no excuse for your not using them. (Even with memory problems, put comments in your final draft and keep a copy.) For example, the comment

```
2150 REM K IS THE SUM OF I AND J
2160 K = I + J
```

is extremely lame. But if, in its place, you write

```
2150 REM K IS SUM OF FIRST AND
     SECOND GROUP SCORES
```

then, within the context of the program, the comment will probably remind you (after a long absence) of several things you'd forgotten. Given the restrictions on variable names in BASIC, comments are more necessary than they would be in languages with longer name possibilities.

I might point out several places where you should always use comment statements. One is at the beginning of the program for a summary of the name of the program, your name as its author, purpose, and so on. See lines 50 to 70 in the NIM program in listing 1 for example.

Another place to put comments is at the beginning and end of modules, if possible, with some eye-catching typography. BASIC programs do seem to run together after a while (see lines 200, 520, 800, 930 and others in listing 1).

A third place to put comments is just before a major control structure (ie: one spanning more than a few lines of code). Gertrude Stein might not have said, "An if is an if is an if. . .," but she should have. Things are easier if you know that an IF statement is actually the beginning of a *do. . .until,* an *if. . .then. . .else,* or something else. For example, look at the comments at lines 810 and 890 of the NIM program:

```
0810 REM DO UNTIL 900; ENDLOOP IF
       VALID (V=1)
(body of do. . .until)
0900 IF V = 0 GOTO 0820
```

A glance at line 810 tells us we are beginning a *do. . .until* that ends at 900; it also tells us the condition and the reason for looping.

Heavily commenting a program reaps such intangible benefits that it is difficult to justify the time, memory and effort that commenting requires. You have always heard that comment lines greatly help a programmer who must examine a program weeks or months after it is written. But you probably do not realize that the very act of writing down the comment, of trying to find the most important few words that will help to clarify the situation, that this very act not only helps you to remember a given fact longer, it also causes you to analyze the given situation (and thereby understand it better), maybe even to find a mistake you had not seen.

Remember, comments may take effort, but the whole idea of structured programming techniques is that effort on the front end will save greater efforts later on.

NIM: Testing (Steps 6 and 7)

A module is tested by writing code around it that provides it with the variables that affect the module's behavior and statements that somehow display the module's output. Then the module-plus-test-routine should be run, varying the inputs across their spectrum as much as is practical (testing all possible input combinations is the only foolproof method but, alas, becomes infeasible very quickly). The outputs should be predicted before the test is run and then verified; "eyeballing" the outputs often lets mistakes slip by that you would otherwise catch.

Program testing is usually more frustrating than module testing, mainly because

Relative Line No	BASIC Line No	Pseudocode
0	—	print-board:
1	1415	print 'THE BOARD IS';
2	1420	sticks = number-of-sticks
3	1430	do while sticks > 5
4	1440	print '/////';
5	1450	sticks = sticks - 5
6	1460	endwhile
7	1470	for I = 1 to sticks
8	1480	print '/';
9	1490	next I
10	1500	print ' '

Figure 7: The print-board *module. This module is actually part of the* evaluate *module but is separated from it for purposes of clarity.* Sticks *is a new variable that is decremented to zero as the current board position is printed; notice the semicolons in the print statements that make the statements print on the same line, as in BASIC.*

Variable Name	*initialize*	*player-turn*	*computer-turn*	*evaluate*	Use
N1					number of games completed; used outside all modules
C	x				user-choice to choose sticks and first player, temporary
C				x	user-choice to resign, temporary
S	x	x	x	x	number-of-sticks
P	x			x	computer's-turn; indicates next to play: computer=1, player=0
V		x			during player's turn, indicates if move valid: 1=yes, 0=no
M1		x			player-move } both = 1, 2 or 3
M9			x		computer-move }
R			x		remainder of number-of-sticks (S) modulo 4, temporary
S1				x	equivalent to S, destroyed by the print-board module

Figure 8: A table to keep track of variables used. This table shows which valid BASIC variable names are being used; in which modules they are used; the variable's meaning and whether or not the variable's value needs to be saved. Note that C *is a temporary variable; since its value in the* initialize *module need not be saved, it is used again in the* evaluate *module for another purpose. In a more complex program, you would make a note by the variable name if it is an array (numeric or character) as opposed to a simple variable.*

Listing 1: The completed NIM game, written in BASIC. This program plays multiple games of NIM against a human opponent with endgame messages to the user that differ from game to game. The two most important characteristics of the program are, first, the liberal use of REM statements, and second, the coding of the program in terms of structured programming control structures, which greatly simplifies program design and debugging. (This program was run on an IBM 5100.)

```
0050 REM ***NIM PROGRAM. WRITTEN BY GREGG WILLIAMS***
0060 REM **WRITTEN 15 APR 77. LAST UPDATE 16 APR 77**
0070 REM **    TRY IT--IT   C A N   BE BEATEN      **
0100 N1=0
0200 REM <<****  MODULE INITIALIZE--END AT 520  ****>>
0210 REM  -GIVE USER INSTRUCTION OPTION IF FIRST GAME (N1=0).-
0220 IF N1≠0 GO TO 0320
0230 PRINT  DO YOU WANT INSTRUCTIONS? (1=YES, 0=NO)'
0240 INPUT N1
0250 IF N1≠0&N1≠1 GOTO 0240
0260 IF N1=0 GOTO 0320
0270 PRINT ''
0280 PRINT 'OKAY.  NIM IS PLAYED WITH 17 OR MORE STICKS. WITH A'
0290 PRINT 'MOVE CONSISTING OF THE PLAYER'S TAKING 1, 2, OR 3'
0300 PRINT  PLAYING PIECES, OR ''STICKS''.  WE ALTERNATE TURNS,'
0310 PRINT  AND THE PLAYER FORCED TO TAKE THE LAST STICK LOSES.'
0320 PRINT ''
0330 PRINT 'I USUALLY PLAY WITH 17 STICKS AND YOU GOING FIRST.'
0340 PRINT ' TYPE 1 IF THAT''S OK WITH YOU, 0 OTHERWISE'
0350 PRINT ''
0360 INPUT C
0370 IF C≠1&C≠0 GOTO 0360
0380 IF C=0 GOTO 0420
0390 S=17
0400 P=0
0410 GOTO 0520
0420 PRINT 'HOW MANY STICKS DO YOU WANT TO START WITH?'
0440 INPUT S
0450 IF S 13 GOTO 0480
0460 PRINT  **SORRY. WE HAVE TO HAVE AT LEAST 13 STICKS**'
0470 GOTO 0440
0480 PRINT ''
0490 PRINT 'TYPE ZERO (0) TO GO FIRST; ELSE TYPE 1'
0500 INPUT P
0510 IF P≠0&P≠1 GOTO 0500
0520 REM  END OF MODULE INITIALIZE-
0590 REM
0600 IF P=1 GOTO 1100
0610 REM
0800 REM   ***  MODULE USER S-TURN--END 930  ***>>
0810 REM  -DO UNTIL 900, ENDLOOP IF MOVE IS VALID (V=1).-
0820 PRINT ''
0830 PRINT 'YOUR TURN--ENTER YOUR MOVE'
0840 INPUT M1
0845 V=1
0850 IF M1<1&M1>3 GOTO 0870
0855 PRINT 'YOUR MOVE ISN''T BETWEEN 1 AND 3'
0860 V=0
0870 IF M1<S GOTO 0890
0875 PRINT  THERE AREN''T THAT MANY PIECES LEFT
0880 V=0
0890 REM <<NEXT STMT IS TEST FOR END OF DO UNTIL LOOP>>
0900 IF V=0 GOTO 0820
0910 REM <<TAKE AWAY FROM CURRENT NUMBER OF STICKS>>
0920 S=S-M1
0930 REM <<END OF MODULE USER S-TURN>>
1000 GOTO 1400
1100 REM <<***  MODULE COMPUTER'S-TURN--END 1240  ***>>
1110 REM <<R IS (#STICKS) MODULO 4>>
1120 R=S-4*INT(S/4)
1130 REM   CASE STMT.--R HAS VALUE 0,1 2,3. SO GO ON R+1>>
1140 GOTO 1150,1170,1170,1190 ON (R+1)
1150 M9=3
1160 GOTO 1210
1170 M9=1
1180 GOTO 1210
1190 M9=2
1200 REM  DECREMENT STICKS AND INFORM USER OF YOUR MOVE
1210 S=S-M9
1220 PRINT ''
1230 PRINT 'MY MOVE IS',M9,'STICK',
1233 IF M9=1 GOTO 1238
1235 PRINT 'S'
1236 GOTO 1240
1238 PRINT ' '
1240 REM   END OF MODULE COMPUTER-MOVE-
1250 REM
1400 REM  <<***  MODULE EVALUATE--END 1770  ***>>
1410 REM  <<PRINT CURRENT BOARD>>
1415 PRINT  THE BOARD IS  ',
1420 S1=S
1430 IF S1<5 GOTO 1470
1440 PRINT  ///// ',
1450 S1=S1-5
1460 GOTO 1430
1470 FOR I=1 TO S1
1480 PRINT '/',
1490 NEXT I
1500 PRINT ' '
1510 REM <<IF #STICKS=1 DO THE FOLLOWING; NEXT STMT STARTS 1580>>
1520 IF S≠1 GOTO 1580
1525 PRINT ''
1530 REM <<NESTED IF--IF COMP''S TURN, RESIGN; ELSE USER LOSES>>
1540 IF P=1 GOTO 1570
1550 PRINT 'OUCH! IT LOOKS LIKE I LOST THAT ONE--NICE GAME.'
1560 GOTO 1580
1570 PRINT 'SORRY, CHUM!  THAT''S THE LAST STRAW--YOU LOSE.'
1580 REM <<NEXT IF DEALS WITH COMP/USER RESIGNATION IF F=6,7,8>>
1590 IF S>8|S=6 GOTO 1770
```

only the most elusive bugs evade module testing. But the method of predicting program behavior and output for a given set of inputs remains much the same as for module testing.

Because I had only four modules and such a simple design, I skipped module tests and went on to test the entire program by playing a few games. I found the following errors: two typing errors, flag V was not set (line 845 was added), and a flag was set wrong at 1540. At this point, the program was functionally working.

NIM: Additions (Step 8)

At this point, the NIM program is finished and running. However, playing several games, I noticed little things that bothered me: lines of output bunching together when they were not logically connected, error messages that needed to be included, the computer writing "MY MOVE IS 1 STICKS," to name a few. So I repaired several things, mostly evident from lines in the BASIC program not ending in zero.

One option that does not show is the *if. . .then* statement at lines 220 and 230 that skips the asking-of-rules (lines 2 thru 6 in figure 3) for every game but the first. Fortunately, this could be added fairly easily by adding a new variable, *games-played* (or N1), updating it (at line 1990), and by placing lines 230 thru 320 in an *if. . .then* structure that gets done only if *games-played* equals zero.

Sometimes it takes more effort to add code so that the resulting program is still structured. But programs resemble organisms in that they tend to grow quite a bit after the first time they are "finished." So, in the interest of maintaining a structured program (which is easier to work on), I make it a rule to add structured code to my programs. In my experience programming at work, it's been worth it.

Final Thoughts

If nothing else, I hope that I've convinced you that time spent in planning is later paid back, and with interest, because that's what structured programming is all about. By planning your program before you write it, you eliminate time wasted in finding out what you've forgotten; by planning your program to fit certain control structures (thereby causing program flow to take a recognizable form), you save time by not having to untangle the spaghetti-like structures that you might otherwise come up with.

Listing 1, continued:

```
1600 IF P=0 GOTO 1660
1610 REM   IF COMP IS TO PLAY, HE MAY OR MAY NOT RESIGN
1620 IF RND(.6 GOTO 1770
1630 PRINT "AGGGH!!  YOU'VE GOT ME. I CAN SEE.  I RESIGN.
1640 S=1
1650 GOTO 1770
1660 IF RND(.3 GOTO 1770
1665 PRINT ""
1670 PRINT "NO MATTER WHAT YOU DO, I'VE GOT YOU.  DO YOU WANT"
1680 PRINT "TO RESIGN GRACEFULLY, OR DO WE FIGHT IT OUT?".
1690 PRINT "  (1 TO RESIGN, 0 TO PLAY)"
1700 INPUT C
1710 IF C≠0&C≠1 GOTO 1700
1720 IF C=0 GOTO 1760
1730 PRINT "OK, I ACCEPT YOUR RESIGNATION.  GOOD GAME."
1740 S=1
1750 GOTO 1770
1760 PRINT "OK, CLOWN, IT'S YOUR FUNERAL"
1770 REM  END OF MODULE EVALUATE
1900 REM   CHANGE P TO REFLECT NEW PLAYER
1910 P=1-P
1920 REM - DON'T LOOP IF END-OF-GAME (GIVEN BY S=1)
1930 IF S=1 GOTO 0600
1940 PRINT ""
1950 PRINT "DO YOU WANT TO PLAY ANOTHER GAME? (1=YES, 0=NO)
1960 INPUT C
1970 IF C≠1&C≠0 GOTO 1960
1980 IF C=0 GOTO 2010
1990 N1=N1+1
2000 GOTO 0200
2010 PRINT "OK.  CALL ME UP WHEN YOU'VE GOT MORE TIME."
2020 END
3000 REM              E N D   O F   P R O G R A M
```

Structured programming in its broadest sense is several things. On the highest level, it is completely knowing the problem. On a middle level, it is the recursive process of repeatedly breaking a problem into subproblems until each subproblem, at whatever level, presents a self-evident solution. (Also, this level requires some awareness of the basic control structures.) On the lowest level, structured programming is writing each subproblem (using one of several given control structures) so that program flow is standardized to one of several recognizable and easily traced patterns.

It's strange that computer programmers took so long to analyze their own programming methods, especially since analysis is so necessary to the problem solving process. But the analysis was finally done, giving birth to the idea of structured programming.

Structured programming is not universally acclaimed. But the fight between pure structured and pure unstructured programming is largely an academic one. In the field, applied structured programming (or many of its techniques, under different names) is essential to programming complex, real world problems. And that means that, even in programs of computer experimenters, it couldn't hurt.■

Decision Tables: How to Plan Your Programs

Thomas G Bohon

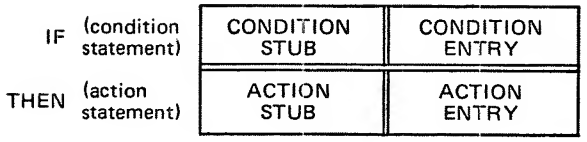


Figure 1: Basic elements of a decision table. A decision table is a formal listing of a series of interconnecting facts and possible alternative actions associated with a particular situation or process.

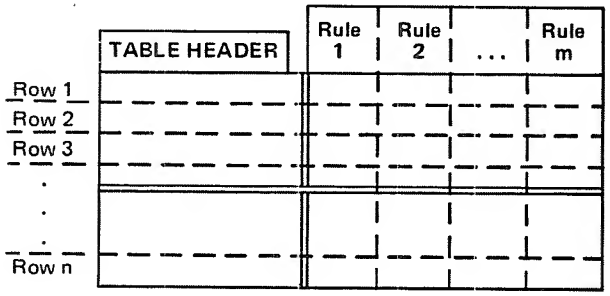


Figure 2: Additional decision table elements. The table header (or name*) allows each table to be uniquely referenced.*

EXTENDED ENTRY EXAMPLE	1	2	3	ELSE
Compare Amount to Discount Amount	Amt 1s Disc	–	Amt Gr Disc	–
Compare Quantity to Quantity on Hand	Qty 1s on Hand	–	Qty Gr on Hand	–
Billing Rate	Regular		Discount	–
Quantity to Ship	Ordered		Ordered	–
Investigate	–		–	ERROR

Figure 3: Example of an extended entry decision table.

MIXED ENTRY EXAMPLE	1	2	3	ELSE
Ordered ⩾ Discount Amount	N	Y	Y	–
Buyer Type	Retail	–	Wholesale	–
Give Discount Billing	–	X	–	–
Back Order Ordered Less on Hand Amount	X	–	X	–
Investigate Error	–	–	–	X

Figure 4: Example of a mixed entry decision table.

"Oh, no," you say to yourself, "Another one of those fancy techniques which no one can understand, I can't use, and I can definitely get along without!"

Did something like the above pass through your mind when you read the title of this article? Well, put aside your doubts for a second and read a bit further. I think you'll be pleasantly surprised to learn that you already know the process I'm going to describe and, in fact, probably use it in a very informal way every day. All I want to do here is to formalize what you already know and show you how you can apply this knowledge to make the job of programming your home computer a little easier.

What am I talking about? Decision tables, of course. And, after reading this article, you should have a better understanding of what they are, how they are constructed, and how to use them effectively.

Some Definitions Before We Begin

A *decision table* is simply a formalized presentation of the mental process each of us goes through every time we are confronted with a series of facts which require us to decide on one course of action or another. Stated another way, a decision table is merely the writing down of the facts and possible alternative actions associated with a particular situation or process.

In programming, decision tables act as effective substitutes for, or as an aid to, the block diagrams associated with preliminary flowcharting. They are used primarily when the situation being studied involves complex decision logic, since the decision table presents not only the original condition but also the course of action in an easy to understand and easy to use tabular form.

There are two main sections of a decision table (see figure 1). The upper section (shown as exactly half of the table, a situation not necessarily found in an actual situation) presents the possible conditions upon which the decision will be based. The lower portion (again, not necessarily half of the table) presents all possible

actions resulting from the possible decisions in the upper portion.

Each portion of the table is further broken up into two sections, with the left hand section being called the stub and the right hand section called the entry. Thus, in our typical decision table we have a *condition stub* and a *condition entry* in the upper portion, and an *action stub* and an *action entry* in the lower portion.

Figure 2 shows the remaining elements of a decision table. Note that there is a *table header* (sometimes called the *label* or *name*) which allows each table to be uniquely referenced. This is necessary in complex situations where the conditions and actions may require multiple tables.

Each *rule* in the entries is identified by a *rule number*. The *condition stub* describes a condition in a way that may be answered either yes or no (in one kind of table) or with a specific value. The *condition entry* provides the means of completing the condition statement. The *action stub* describes the action(s) to be taken, while the *action entry* provides the means of showing completion of the actions.

Decision tables are generally classified by the type of information recorded in the entries. There are three types generally accepted:

- **Limited Entry**: This is the most widely used and, because of its similarity to binary logic, is most suited for computer oriented applications. Condition entries are limited to a Y, N or – (meaning not applicable). Action entries are limited to Xs. In order to accomplish this, the condition stub must be written so that a true-false condition exists, and the action stub must describe the complete action to be taken. The example in this article will be of this type.
- **Extended Entry**: In this type of decision table, the entry portion is merely an extension of the stub portion. The stub describes the variable and the entry describes the possible values which the variable can assume. This type of table is quite well-suited for those situations in which only a few variables occur, except that those few variables may assume many different values. Figure 3 is an example of this type of table.

Note: For those readers who would like to learn more about decision tables, I recommend the following books:

Automatic Data Processing: Principles and Procedures by E Awad and DPMA

Decision Tables and Their Practical Application in Data Processing by Thomas Gildersleeve.

Both of these books are published by Prentice-Hall, 1970. I would also be happy to answer any questions raised by my article.

Closed Table Example #1	1	2	3
condition			
condition			
action			
action			
DO 2	X		
action			
DO 3		X	
action			

Closed Table Example #2	1	2
condition		
condition		
action		
action		
RETURN	X	X

Closed Table Example #3	1	2	3
condition			
condition			
condition			
action			
action			
EXIT	X	X	X

Open Table Example #1	1	2	3
condition			
condition			
action			
action			
action			
GO TO 2	X	X	X

Open Table Example #2	1	2
condition		
condition		
action		
action		

Figure 5: Examples of open and closed decision tables. An open table has as its last action in each rule a branch to the next table in the series. Closed tables return control to the tables that call them upon completion of their routines.

- Mixed Entry: As the name implies, this type of decision table has rows which contain either limited or extended entries. See figure 4 for an example.

As mentioned above, complex situations may require more than one table, or you may place different types of decisions in different tables. Obviously, there must be a way for one table to reference another and indeed there is. How? Simply by the type of table you build. An *open table* has as its last action in each rule a branch to the next table in the series. This transfer is a permanent one and is accomplished by an action stub of GO TO n. A *closed table*, on the other hand, uses an action stub of DO n or PERFORM n with the idea that, after the called table is completed, control will return to the calling table and the indicated actions from that point on will continue. Return from the called table is through an EXIT or RETURN action entry. Figure 5 gives examples of both open and closed decision tables.

How to Construct a Decision Table

The first step in constructing an effective decision table is to state the problem in a clear and concise manner. For example, suppose we wish to construct a table for the following hypothetical situation:

> *Your firm, which manufactures fridgets for home computers, often sells on credit. If a customer places an order which exceeds his/her previously established limit, the order should be forwarded to the credit manager for approval prior to filling and shipping it. However, if the customer has purchased more than $600 in the past six months, he/she is considered a regular customer, and in such cases tentative approval is assumed and the order is filled but not shipped until credit appoval is received. There is also a minimum order value of $100 from all customers and all orders less than this amount must be returned unfilled unless the order is from a regular customer in which case it may be filled and shipped. All orders over $500 in value receive a 10% discount and all orders over $750 receive an additional 5% discount. However, the discounts apply only for regular customers as defined above.*

By stating the situation as we have, we have completed the first step in our decision table construction (I realize that I said the statement should be clear and concise, but we have to have something to work with!).

The second step in our construction process is to isolate and list both the conditions which will affect our eventual decision and the possible actions we may take:

Conditions	Actions
Regular customer	Request credit approval
Order exceeds credit limit	Fill the order
Order less than minimum	Ship the order
Order less than $500	Reject the order
Order is over $500, less than $750	Give 10% discount
	Give 15% discount
Order is over $750	No discount

At this point we should stop and examine our lists for correctness and add any items which have been omitted. In our example, the last three items in each list are redundant: obviously, a single order cannot possibly require all three checks nor is it necessary to keep all three actions. It would be much simpler to check each order for "over $500" and "over $750," assuming that the only possible other condition will be "under $500." Similarly, instead of listing all three discount possibilities, why not list "give 10%" and "give an additional 5%" — this covers all possibilities. After our examination and the elimination of these redundant conditions, we have the following revised lists *[Note: There is no implied relationship between Conditions and Actions at this point]:*

Conditions	Actions
Regular customer	Request credit approval
Order exceeds credit limit	Reject the order
Order less than minimum	Give 10% discount
Order over $500	Give additional 5% discount
Order over $750	Fill the order
	Ship the order

The next step is to place these conditions and actions into a formal table structure. A general rule to follow when constructing the actual table is to list the actions in the order in which they are to be performed. Further, a condition entry is left blank (not applicable) only if the condition is either not possible or is overshadowed by other conditions also present. Our table, in skeleton form, appears as in figure 6.

After we have filled out the condition and action stubs of our table, we must complete the entry portion by filling in the rules. This is accomplished by returning to the original problem statement and carefully marking the condition entries and the associated action entries. This is shown in figure 7a.

The final step in building our decision table is to insure completeness and eliminate both redundancy and contradiction. Contradiction is best eliminated by careful

Sample Table	1	2	3	4	5	6	7	8	9	10
Regular customer										
Order exceeds credit limit										
Credit approval received										
Order less than minimum amount										
Order > $500, less than $750										
Order > $750										
Request credit approval										
Give 10% discount										
Give additional 5% discount										
Fill the order										
Ship the order										
Reject the order										
Investigate error										

Figure 6: A preliminary decision table based on the example in text.

Sample Table	1	2	3	4	5	6	7	8	9	10
Regular customer	Y	Y	Y	Y	Y	N	N	N	N	
Order exceeds credit limit	Y	Y	N	N	N	Y	Y	N	N	E
Credit approval received	N	Y	–	–	–	Y	N	–	–	L
Order less than minimum amount	–	–	Y	N	N	–	–	Y	N	S
Order > $500, less than $750	–	–	–	Y	N	–	–	–	–	E
Order > $750	–	–	–	–	Y	–	–	–	–	
Request credit approval	X	–	–	–	–	–	X	–	–	–
Give 10% discount	–	–	–	X	X	–	–	–	–	–
Give additional 5% discount	–	–	–	–	X	–	–	–	–	–
Fill the order	X	X	X	X	X	X	–	–	X	–
Ship the order	–	X	X	X	X	X	–	–	X	–
Reject the order	–	–	–	–	–	–	–	X	–	–
Investigate error	–	–	–	–	–	–	–	–	–	X

Figure 7a: A skeleton decision table developed from the preliminary table in figure 6.

Corrected Table	1	(2 & 6)	3	4	5	7	8	9	10
Regular customer	Y	–	Y	Y	Y	N	N	N	
Order exceeds credit limit	Y	Y	N	N	N	Y	N	N	E
Credit approval received	N	Y	–	–	–	N	–	–	L
Order less than minimum amount	–	–	Y	N	N	–	Y	N	S
Order > $500, less than $750	–	–	–	Y	N	–	–	–	E
Order > $750	–	–	–	–	Y	–	–	–	
Request credit approval	X	–	–	–	–	X	–	–	–
Give 10% discount	–	–	–	X	X	–	–	–	–
Give additional 5% discount	–	–	–	–	X	–	–	–	–
Fill the order	X	X	X	X	X	–	–	X	–
Ship the order	–	X	X	X	X	–	–	X	–
Reject the order	–	–	–	–	–	–	X	–	–
Investigate error	–	–	–	–	–	–	–	–	X

Figure 7b: The final corrected decision table for the example in text.

examination of the problem statement to insure that the conditions and actions we entered into the table earlier do not contradict each other. Insuring completeness is fairly simple if we understand the "else rule." Put simply, this rule says that, if none of the other rules listed hold, we also have a specific action to take. In the case of our table in figure 7a, the "else rule" says we are to investigate the error condition.

Redundancy

Eliminating redundancy is a bit more complicated. There are various rules and methods for doing this, and we will discuss only one of them. Certainly this is not the only "right" method. Also, keep in mind that throughout the following discussion we are dealing only with two rules which have the same indicated actions.

The first law for eliminating redundance says:

> *If, with the exception of one condition, two rules have the same condition entries and, for that one condition, one rule has a Y entry and the other an N entry, then the two rules can be combined into one rule with the entry for that condition becoming indifferent (not applicable).*

Let's apply this law to our table in figure 7a. Note that rules 2 and 6 seem to fit the criteria: they both have the same action entries and the same condition entries, making them candidates for elimination. We can thus combine these two rules with the result shown in figure 7b. Note that rules 3 and 9 almost fit our criteria for possible elimination: the only difference is that there are two conditions with different entries, and we are allowed only one by our rule. No other rule pairs fit the criteria and, after combining the two rules as in figure 7b, we may safely assume that our table passes this first law of redundancy elimination processing.

The next test to apply can be stated as follows:

> *Each pair of rules remaining after application of the test above must have at least one condition for which one rule has a Y entry and the other an N entry.*

Those pairs of rules which meet this test are said to be *independent* of each other, while those which fail this test are said to be *dependent* on each other. Dependency at this point in our tests indicates that the table still contains either redundancy (it has a dependent rule pair with the same actions) or contradiction (there is a dependent rule pair with different actions). Let's examine our table.

Pairing each rule with each of the others, one at a time (eg: pair 1 and 2, then 1 and 3, 2 and 3, 3 and 4, and so on), we check the conditions for a Y in one rule and an N in the other. This isn't as time-consuming as it appears, since we can assume the pair is independent upon encountering the first occurrence of the Y-N condition. We can see, after examining all rule pairs, that none of them are dependent. We can therefore assume that our table is indeed nonredundant and that it does not contain any contradictions.

Note: *If we had found a dependent rule pair, we would have had to apply the following rules to eliminate the redundancy:*

1. *If one rule is pure and the other mixed, then the pure rule is contained in the mixed rule and the pure rule may be eliminated. (A* pure *rule is one in which all entries are either Y or N, while a* mixed *rule has both Y and N entries.)*
2. *If both rules are mixed, there is at least one pure rule which is common to both which you can eliminate from one of the original rules.*

We won't go into these here, since they usually appear only in more complicated applications. I mention them simply to make our discussion complete.

Once our decision table is built and we have completed the error checking procedures mentioned above, we can use the table as a basis for either a preliminary flowchart or, with the addition of the necessary IO routines, go directly to the coding phase of our programming. The path we take at this point depends entirely on how carefully we have constructed our table.

Conclusion

We have seen how we can go from a generalized problem statement to a list of possible conditions and actions to a completely checked out and (we hope) error-free decision table. Of course, like any other new procedure, you will have to use it several times before you become comfortable with the process. But no matter how difficult or complicated it seems, I urge you to try it not once but several times in actual programming situations. After doing so, I'm sure you'll agree that using decision tables greatly increases your productivity and eliminates the situation in which, almost at the end of a long program, you discover one little condition you forgot back at the beginning, which is where you end up again in short order! ■

Programming Entomology

Gary McGath

An entomologist is a bug expert. When he sees an insect, it isn't just a bug to him (in fact, he will vociferously protest that not all insects are bugs); it has a particular habitat, lifespan, favorite food, and breeding pattern. Nor is his knowledge just academic; he can tell you how to protect yourself from a harmful one by killing it or keeping it away.

The same sort of knowledge is necessary for programming. The skilled programmer knows what kinds of bugs may attack a program, how to track them down, and how to keep them from getting there in the first place. He knows the ways to get at particular bugs, as well as the general treatments which are effective against all of them.

The first thing to realize about bugs is that they don't appear by spontaneous generation. They have a creator, and their creator is the programmer. (Throughout this article, I am speaking only of user program bugs; hardware bugs are an entirely different breed, subject to different laws, and systems software may be beyond your control.) No matter how outrageously the program is acting, it's only following orders. So what you have to ask about a bug in your program is: how did *you* put it there? What kinds of mistakes are you prone to make? If you caught a certain bug in one part of the program, might you have put the same kind of bug elsewhere as well? "Thou art God" . . .and thou must take care of thy creation.

But the fact that each programmer creates his own bugs doesn't mean there aren't species of bugs found in everyone's programs. Knowing about these species can be a great timesaver, especially when the species can be identified by the effects.

One of the most common bugs is the Clobbered Value, found where the programmer assumes the content of a register or the value of a variable is the same as before, but it isn't. Take this attempt to exchange the values of two variables:

```
10 LET X = Y
20 LET Y = X
```

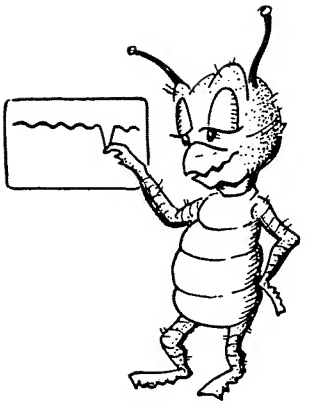
This fails because when statement 20 is executed, the value of X has already been clobbered by the previous statement, with the result that Y never gets changed at all.

Clobbered Values are frequently found on subroutine exits. It's easy to write a harmless looking CALL or GOSUB (possibly to a routine you haven't written yet) and assume everything will remain the same. But strange things can happen if the subroutine unexpectedly changes some values.

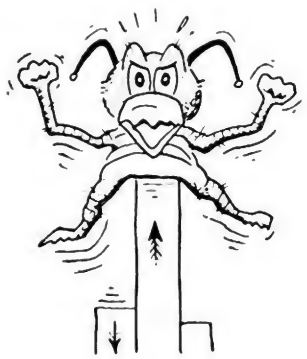
A not too distant relative of the Clobbered Value is the Zapped Stack, found only in machine and assembly code. It appears most often by pushing items onto the program's stack at the start of a subroutine, then failing to pop them, or popping too many things at the end. Another way to invite this bug is to use the stack pointer for some other purpose during the course of a subroutine.

Subroutines are also the habitat of the Botched Call. A certain protocol is needed to call any particular subroutine. If, when you write a call to a subroutine, you expect a value to be returned in the wrong place, or you assume the subroutine will do something which it actually won't (or vice versa), this bug will have gained a foothold. The difference between a Clobbered Value and a Botched Call is that when you have the latter, the subroutine is doing the right thing; the calling program is just mistaken in its expectations.

Another species of bug lurks in jumps, branches, and GOTOs. The Branch Bug is so difficult to fight that serious attempts have been made to wipe out its habitat; languages and programming styles (structured programming) have been developed that use no jumps. The Branch Bug comes in two varieties: jumping to the wrong place, and jumping to the right place with inadequate preparation. The first of these is easy to produce in languages where statement labels have to be numbers (eg: BASIC and FORTRAN, especially BASIC, where every statement has to be numbered whether it's ever going to be a jump destination or not). The jump with inadequate preparation is similar to the Botched Call,



Clobbered Value Bug: Your program changes the value of a variable at a time and place which is unintended. The detection difficulty ranges from the obvious (after if is found) to the subtle (before it is found).



Zapped Stack Bug: Stack oriented machines and software are both very egalitarian with respect to pushes and pops. They like to have the same number of items pushed as are later popped, or else they'll transform themselves from tranquil and placid programs into memory zapping monsters.

but it can often be harder to figure out if the program has a complex flow pattern.

A few special methods are applicable to fighting the Branch Bug. One of these is program flow analysis. A look at the possible paths a program can take will often reveal some of these bugs. Is there a part of the program that can never be reached? Are there traps in the program, loops that can never terminate? Are there jumps which will result in variables being used without having been set to a value?

In languages like BASIC, where every statement is labeled, it's helpful to set off statements that can be reached by jumps either by using special statement numbers or by pointing them out in comment statements. In any language, the statements that can be reached by jumps should be logical breaking points in some sense, places where a new unit of work begins. Except in desperate situations where economy is all-important, jumps should be used to satisfy the logic of the program, not to save a few instructions.

If a subroutine call can be used instead of a jump, it probably should be used. A subroutine will send you back where you came from, so figuring out the flow of the program is easier. For many purposes, you can treat a subroutine as a unit when studying the program; as a single instruction that happens to do complicated things. You can't do this with the instructions reached by a jump.

The next bug in our survey feeds on apples and oranges. More generally speaking, the Mismatched Unit is found where the units or dimensions of the quantities being used in a program aren't the ones actually needed. Take the program statement LET V = D * T, where D is a distance in miles, T is the time traveled in hours, and V is intended to be the traveler's average velocity in miles per hour. By using simple algebra on the units, you can see that the result obtained will be units of miles *times* hours, not miles per (ie: divided by) hour.

Botched Call Bug: The Botched Call Bug is like the proverbial square peg in a round hole: Unless the peg or the edge of the hole yields, sparks will fly.

Bugs of this type are harder to spot when the mismatched variables are further apart in the program, but consistency will keep them from occurring. Simply be sure you know in advance what units each variable has to come in.

Assembly and machine language programming allow an especially messy type of Mismatched Unit to show up: mismatches between addresses and data, or between absolute addresses and relative addresses (values to be added to a base address). To avoid this bug, watch out for the different addressing modes of different instructions.

Another bug with a specialized habitat is the Fencepost Bug, named for its tendency to rest in problems like this one: "If you are putting up a wire fence 100 feet long, supported by posts every 10 feet, how many posts do you need?" Another name for this bug is the Boundary Condition Bug; it's always found in connection with the start or end of some sequence, where special treatment is needed. One form manifests itself in confusion over whether the first element of a group is number 0 or number 1. Another is found in the attempt to relate each element of an array to the next, as in this statement:

IF T(I) <T(I+1) GO TO 100

Try this one setting I equal to the dimension of T.

Finally, we come to the most insidious of all bugs, the Timing Bug. The characteristic that makes this bug so fearsome is that a program infested by one may run correctly once but not the next time; it may even run correctly 99 times but fail on the hundredth, using exactly the same data each time. To make matters worse, running programs in single step mode will usually drive Timing Bugs into undetectable hiding.

As the name suggests, the Timing Bug is one that shows up depending on the order in which asynchronous events (events that have an unpredictable relationship in time)

Mismatched Unit Bug: A result of inadequate analysis of a calculation, the Mismatched Unit Bug results in strange elixirs. When both apples and oranges are thrown into the analytical engine, what is the nature of the juice which flows out?

occur. Systems that have interrupt facilities are especially prone to being attacked by Timing Bugs, since an interrupt routine may be executed at a different point in the program each time it's run. An interrupt routine may, for instance, set up certain variables to be used by the main program. If another interrupt of the same kind can occur before the variables have been processed by the main program, and if that interrupt changes those variables, unpredictable results can occur. Yet most of the time, interrupts may not occur that close together, so the bad result is said to be nonrepeatable. This means that repeated runs of the program can't be used to systematically close in on the bug.

A Timing Bug can also live on direct memory access (DMA). Some mass storage devices can read or write data in bulk without the intervention of the processor, using those memory access cycles which the processor doesn't use. The length of time a DMA transfer will take is, at best, very difficult to predict; so a Timing Bug can strike if memory which is accessed by DMA can be accessed or modified by the processor.

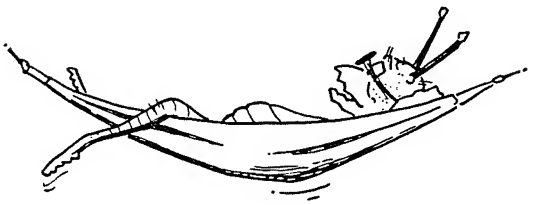
Since Timing Bugs are so hard to hunt down, extra efforts should be made to avoid giving them a foothold. Be extra careful in writing interrupt handlers or DMA commands. Watch for places where interrupts need to be disabled. As for the indentification of Timing Bugs, the following rule is useful: if you can prove, in a precise instruction by instruction study, that what happened couldn't possibly have happened from the execution of those instructions, suspect a Timing Bug; something else was happening during the execution of those instructions.

Incidentally, it's possible to encounter bugs much like Timing Bugs even without interrupts or DMA. An input or output device, such as a keyboard, is asynchronous with the program; the exact behavior of the program will depend on the behavior of these devices. For instance, a program which accepts keyboard input and accumulates it in a buffer may work fine for you, yet a faster typist may make it fail because no provision was made for the chance of exceeding the buffer's capacity. But in a situation like this, it's at least possible to look at every call to an input routine and tell what its effects might be.

This completes our survey of important species of bugs (I have nothing useful to say about the Common Typo, though it does have to be fought). Others will no doubt discover voracious breeds which I have overlooked, and perhaps they will improve on some of the classifications I have mentioned. But knowing about the species which are listed here will hopefully be a help in identifying and killing the bugs in your own programs.

Branch Bug: Jumping blindly about in memory, the Branch Bug is always on a collision course with valid execution of a program.

The Timing Bug: This most subtle of all bugs spends most of its time relaxing, and suddenly taking a swipe at apparently random times.



This doesn't mean that classifying bugs is all there is to entomology, neither the biological kind nor the kind being discussed here. Entomology wouldn't be a science if it couldn't say things that are true of all bugs, regardless of species. What I have discussed so far is differentiation; but integration is equally important.

The basic fact that unifies all bugs is the one which I mentioned at the beginning of this article: they're all creations of the programmer. And this fact allows the use of a broad-spectrum killer against all bugs: DDT, standing for Design, Documentation, and Testing. Let's take them in order:

● **Design.** The best way to stay bug-free is to write programs without bugs. This may sound like superfluous advice, but programmers (myself included) are often tempted into writing programs quickly, rather than writing them well. The attempt usually fails, since such programs will usually cost more in debugging time than the time saved in writing them.

An error born of pragmatism is to suppose that it doesn't matter how you design a program, as long as it works. There are two problems with this idea. The first is that if you use any method that appears to do the job, without regard for well organized design, it will be a lot harder to ever make the program work. The second problem is that even if the program works for its immediate purpose, it will be harder to make changes to meet new needs, since a particular ad hoc solution may not be generalizable.

The first step in designing a program is to lay out a complete plan of attack before writing it. Decide what data structures you will need, and what method you will use. Data structures are often the key to the whole program. First plan the program in a few large steps; then decide what each step will consist of in more specific terms; then repeat the procedure until you're down to the level of your chosen programming language. This is the principle of structured programming, and also of mental unit-economy: avoid having to think about more things at once than your mind can handle. If you can keep everything relevant to a particular operation in your head, you're not likely to put bugs into its implementation.

Flowcharting is often recommended for program design, but it's cumbersome and doesn't lend itself to representing a hierarchical design. Another approach is to use a well designed programming language, such as ALGOL or APL, to write the design. Since you aren't actually going to run the program in that language, you can assume any features that would make the job easier. The point of this is to have a representation of the program that you can understand without strain, so that you don't lose sight of your overall plan while chasing down details of implementation. If you do have bugs after doing this, at least they won't be part of the whole design of the program.

● **Documentation.** The main reason for writing up the way a program works isn't to explain it to someone else; it's to make sure you understand it yourself. Documentation shouldn't be an afterthought; it should begin with the design of the program (when you write what it is going to do), and continue with comments written along with the instructions.

Good documentation isn't found in sheer number of comments (though there should be a lot); it's found in comments that explain the operation of the program. Comments are especially needed for data, subroutines, and points reachable by jumps. Variables and constants should be explained so that the reader will see how they can be used; this allows us to spot threats to them, such as Mismatched Units and Clobbered Values. If the language allows, give constants names rather than using their numeric values throughout the program; this makes updating easier and renders the Common Typo's attacks more conspicuous. Subroutines should be prefaced with a description of how they are called, what inputs are needed, what values are returned, and what information may be destroyed in the process. Jump points should have an explanation of the conditions under which they are reached.

To make a program at least partly self-documenting, the name of a routine or variable should indicate its use. One of the major weaknesses of BASIC is that it doesn't allow this to be done very much; this is a reason for having a lot of comment statements to explain what BASIC variables and subroutines are used for.

Just as a sample, here's a preface to a hypothetical 8080 assembly language subroutine (see box). The comments explicitly define linkage conventions.

The protection provided against Botched Calls should be obvious.

● **Testing.** If you follow the approach outlined so far, you'll have a better chance of getting your program to work, but you may still have planted a few bugs inadvertently. So you have to test the program before declaring it bug-free. Testing should begin with a simple version of the program, if possible; but it should begin only after the program has been written with enough care so that there's a chance

of not finding any bugs.

Use whatever debugging tools are available. High-level languages will usually provide useful information when the program goes wrong. Versions of BASIC that allow single statements to be executed make it possible to find something about the conditions under which an error occurred.

When working in machine language, a debugging program will ease discovery of bugs. Such a program allows the user to put breakpoints into the program being tested (returning control to the debugger when the program counter reaches a certain address) and to examine and modify registers and memory. These programs range from simple 1 K monitors to powerful symbolic debuggers like Digital Equipment Corporation's DDT (Dynamic Debugging Tool, no relation to the name as used here). Having one of these in ROM can be a tremendous help.

If the program works the first time, try it again with different data to make sure. Check out simple cases. Sometimes a program will work in complicated cases, but be bitten by the Fencepost Bug in simple ones. Check out more complicated cases. If possible, use a random number table as a source of test data, along with handpicked cases.

If the program *doesn't* work the first time, try it again with different data. Aim for the simplest case possible. If you can get the program to do something right, that will cut down the number of places where bugs may be lurking.

When a program is being tested, the work is easiest if execution comes to a screeching halt as soon as something goes wrong. A program may be able to run a while after crucial damage has occurred, only to clobber all of memory before stopping. If this happens, it can be almost impossible to localize the source of the disaster. But if the program makes periodic checks for error conditions (such as impossible values or invalid relationships) and reports them, there's a better chance of discovering just where things went wrong. For instance, a routine that fills a block of memory between two addresses might check to make sure that the low address is really lower than the high address. Redundant tests may slow down the program, but they can be taken out when all the bugs are known to be dead.

The overriding consideration to remember in the use of this Design, Document and Test technique is that it's open-ended. It will, in principle, kill any kind of bug; but a new approach to design, a better scheme of documentation, or a novel test may be needed for subtle species. Approaching bugs scientifically means thinking about them. It means recognizing that any bug will have important similarities to previously encountered bugs; and that it may have equally important differences. So when you find yourself struggling to discover what's wrong with a program whose behavior is incomprehensible, you can console yourself with the thought that you may be about to make an exciting entomological discovery that you can use repeatedly.■

```
;  COMPUTE PROBABILITY OF WIDGET BREAKAGE
;  INPUT – MASS OF WIDGET (GRAMS) IN REGISTER PAIR BC
;          AGE OF WIDGET (DAYS) IN REGISTER PAIR DE
;  OUTPUT – PROBABILITY OF BREAKAGE (PERCENT) IN REGISTER PAIR BC
;  ALL OTHER REGISTERS ARE CLOBBERED
```

PROGRAM DETAILS

About This Section

This section deals mainly with one of the more difficult aspects of a program's structure tables. For any but the most elementary applications the programmer finds that he (she) needs to construct some kind of table for a variety of purposes: branching, symbols, data. In fact, note that virtually any file of data can ultimately be thought of as a table. This section should answer many of your questions about a variety of tables.

The second topic covered in this section is how to create and maintain binary trees. This subject has a reputation which scares a lot of people from using trees. But when working with large amounts of unsorted data, many times the fastest way to reference any particular piece of it is by arranging it using a binary tree approach. Now there is no longer anything to fear about binary trees.

An Introduction to Tables

F James Butterfield

The construction and use of program tables is the gateway to developing powerful programs. The new programmer may have trouble getting to know the concept of tables, but time spent learning about tables is well worth the effort.

The first few programs to go into your home computer are likely to be written using a multitude of IF tests: If a value equals 1, branch to a particular routine; if equal to 2, another branch; if over 5, yet another branch; and so on. After a while this gets to be a lot of work. Programmers quickly learn to use table structures to simplify decision making.

Tables are called by many names, depending on the language and the application: arrays, vectors and matrices, to name three. Even the concept of a "file" is usually just a large table which follows the same structural rules but is stored on disk or tape.

Table Elements

Most of the tables we meet in books, forms and so on consist of data arranged in rows and columns. Each row usually contains a record about something. Name, address, age, phone number might be the record of a schoolmate. Each item of this record, such as name, is called a field. In most cases, each record contains the same number of fields; this is called a rectangular table because of its appearance when printed, and is by far the easiest type to handle.

Rows and columns can be interchanged, of course, by laying the table on its side. Let's look at two ways to encode this small table:

Name	Age	Phone
Joe	14	515-3838
John	18	216-3001
Pete	17	414-3377

First we could encode each line this way:

record 1	field 1	Joe
	field 2	14
	field 3	5153838

This is the most common, and usually the handiest way to set up the table. It's logical, easy to change or to add new items, and not difficult to program a search routine for. All the data for a particular line of the original table is in one record. However, during this search, we must leap 12 bytes or so each time we wish to examine a new record. This may or may not be convenient to do, depending on hardware characteristics. By laying the table on its side, we could write:

record 1	field 1	Joe
	field 2	John
	field 3	Pete
record 2	field 1	14
	field 2	18
	. . . etc	

This method is in some ways like devoting a separate table to each kind of data in the big table: a table of names, a table of ages, etc. This type of organization might make it a little easier to search for a name, but it becomes tougher to add a new name to the list, and harder to read. But either way works.

Order of Items

One of the most important decisions you must make in designing a table is how to order the records. For small tables it doesn't matter very much. But as tables get bigger, it becomes important not to waste time on lengthy searches.

At first glance, the simple answer is to put the most often used items at the top of the table where they'll be found first, a procedure which frequently works well. But you must know roughly how often each table item is likely to be used. If the usage pattern changes, your table lookup becomes inefficient. Beware of elaborate schemes to

rearrange the table order as usage changes: they can quickly use up more time than they save.

An excellent method for ordering tables is to use the table address itself as the item to be matched. Let's clarify this with an example. Suppose we have a character in Baudot (5 level) code that we want to translate, say, to ASCII. The lowest value possible is blank, or 00000 (decimal zero). The highest value is the letters shift, or binary 11111 (decimal 31). If we add this character, as a binary number, to the table base address, we'll create an address ranging from TABLE+0 to TABLE+31. In each of these table locations, the corresponding ASCII character will be stored. We'd have to make provision for both upper case and lower case Baudot, of course. The important thing about this kind of table is that we never have to search it. We go straight to the address we want.

The most common way of ordering items in a table is sequential, ie: in ascending or descending order, alphabetically or numerically. Usually we must pick one particular field for the sequence, the one we expect to search most often.

We get many advantages when we have a sequential table. The program can detect right away if it has "gone past" the item it's looking for, so that it won't waste time searching through the rest of the records. With a little more programming effort, we can write a binary search program that passes through a table very quickly. The binary search routine works by examining the middle of the table and deciding if the desired item is above or below this point. From then on, the program concentrates exclusively on the remaining half of the table, and looks at its midpoint in the same way. Each step cuts the remaining portion of the table in half; eventually the desired location is found or a conclusion of "no match" results.

A sequential table is the only type that can be used for a continuous value calculation. You may recognize the following partial table·

Income	Tax
less than 2350	0
less than 2375	2
less than 2400	5
•	
•	
•	

This table associates a continuous value, income, with unique tax amounts. If your income was $2378.54 you do not escape tax because there isn't an exact value of $2378.54 in the table. For your program to find such an intermediate value, the table must be sequential.

There are several drawbacks to sequential tables. The first is the problem of getting the table in sequential order and keeping it that way during deletions and additions. The second is that only one field is in sequence. This means that the user may have to re-sort the whole table to start searching on a new field.

Advanced Techniques

When it is desired to arrange a table in some order, there may be some difficulty moving the items around, especially if they are large and clumsy.

One way to get around this is to leave the data in its original order and build a separate table called an index which gives the order in which the data should be read. This way, instead of moving the data around, the index is simply changed as necessary.

Another way to achieve a similar effect is by chaining. This attaches an extra field to each record which points to the record to be looked at next. The program must have a starting point that tells which record is to be examined first. From then on, the program follows the chain to the last record.

Indexing and chaining are both relatively complex, but they have one important advantage: the same file can have two indices or two chains so that it is simultaneously sorted two different ways. This feature can sometimes eliminate many time-consuming sorts.

Tables which are not rectangular are a source of difficulty. If we are recording, for example, names of parents and their children, we soon face the problem of some parents having only one child, while others have seven or more. Should we allow seven slots for each set of parents and waste precious memory? We could build a complex table structure to allow for a variable number of fields (children). This is practical, of course, but sometimes we can eliminate the problem by making the table into a list of the children rather than the parents.

Another special case which is often encountered is the triangular table, which resembles a square split along the diagonal, with the two halves containing the same numbers. For example, if you calculate a table of mileages between cities, you don't need to store both the Buffalo to Denver and the Denver to Buffalo mileages; they are of course the same. But trying to store only half the table to save memory turns out to be a difficult task. You'll need a medium sized program to get to the right spot in the table.

Access

The addressing modes of your machine warrant study to determine the best way to scan tables. If you have a hardware index register, that's usually the best way both in terms of speed and programming convenience. Each microprocessor has its idiosyncrasies. An 8 bit index will only cover a table size of 256 locations. Sometimes, though, an index doesn't modify a full address, but only an 8 bit offset. In this case the index must hold a full address rather than a simple table position. How easy is the index to modify as you step through the table? An increment command that adds one to the index value is of limited value if you want to jump 12 locations at a time.

If indexing isn't convenient for a given job, indirect addressing is the next best bet. Put the address of the start of your table into an indirect address location; then add to it as necessary until you reach the end of the table.

Don't hestitate to search a table backwards if it's convenient. This facilitates searches when using certain types of indexing.

Program Intercommunication

One program segment can communicate with another by means of tables. In fact, processors which feature a common memory use this technique. When working with an interrupt structure, the recommended procedure is to have one program prepare a table of material for another to pick up. This becomes a good way to segment large projects into convenient modules. Each module can be separately debugged by preparing a set of test input tables and examining the output tables it produces. On very large jobs, this kind of segmentation is an excellent way to divide work among several people. Even online debugging becomes easier, since the tables can be readily viewed at any time.

Conclusion

Tables are a good way to arrange data in a compact, visible and easy to modify form. New programmers sometimes have problems getting used to designing and using them, but they are well worth the effort.■

Hashed Symbol Table

Hashing is the meat and potatoes of symbol table handling.

John Beetem

It is often necessary to convert alphanumeric code into numeric code efficiently. This article describes how to do this using a powerful data structure called the hashed symbol table. Assembly language code is included for the 8080 microprocessor, but the algorithms and structures apply to any computer.

A symbol table is a set of ordered pairs called entries. The first element of each pair contains a symbol (usually in ASCII) and the second contains the object the symbol represents. There are three operations which are applied to a symbol table:

- **Lookup** (also called search): An input symbol, called the key, is compared to the symbol in an entry of the symbol table. When a match occurs, the object associated with the symbol is output. If no match occurs, this condition is indicated;
- **Insert**: An entry is appended to the symbol table;
- **Delete**: An entry is removed from the symbol table.

A structure to make these operations easy and efficient is the object of this article.

Lookup

Lookup, if done wrong, can be a very time consuming operation. The most fundamental lookup structure is a simple array where the entries are placed sequentially in memory. If the number of entries is large, lookup is quite slow because the key must be compared to half of the entries on the average. A sorted array of entries can be searched by methods such as a binary search, which is considerably better (and much more complicated.) But the best method seems to be one called hashing.

A hashed symbol table consists of many arrays of entries, called buckets (my system uses 64 arrays). Each element in a bucket has the same hash code for its symbol. A hash code is computed from the symbol itself using a pseudo random method, such as adding the binary representations of all the characters in the symbol and using the low order six bits of the result. Using a good hashing method, the symbols are well distributed over the buckets, and each bucket is fairly short.

A Note About Notation

*The routines described in this article are represented in two notations. Figures 1 through 5 show the various algorithms in the Warnier-Orr structured programming discipline. This notation is more fully described in David Higgin's articles in the PROGRAM STRUCTURE section of this edition. Listings 1 through 5 provide the author's corresponding 8080 assembly language versions of the programs. . .***BL**

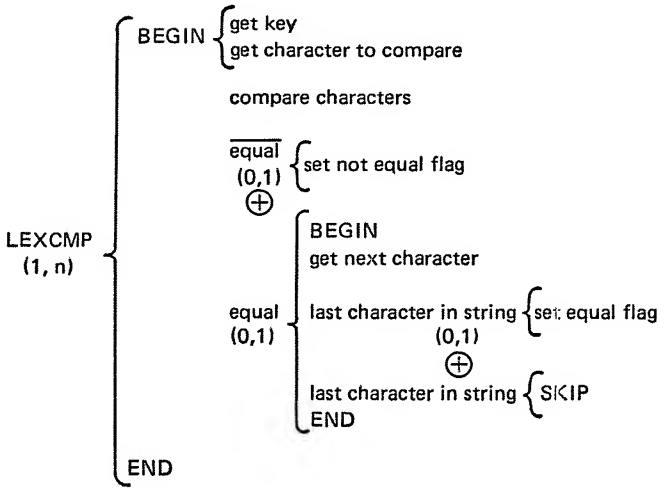


Figure 1: LEXCMP compares two ASCII strings. My format assumes that the last byte of each string is marked by its sign bit being set. The strings can be of any length, and don't have to both be the same length. Equality of strings is indicated by returning with the match flag set. Otherwise the match flag is cleared.

To lookup a key, the following algorithm is used:

- Compute the hash code;
- Use the code to find the bucket;
- Use a simple sequential search through the bucket to match the key.

Since each bucket is short, this is an efficient way to perform the lookup.

To insert or delete an entry, first hash the symbol to find the right bucket, then insert or delete the entry into or from that bucket. This last operation is dependent on bucket structure, and will be discussed presently.

Storage could be a problem. Would it make sense to have 64 different arrays, one for each bucket? No, because one bucket could become filled while others are empty, and it's silly to run out of space when there is plenty left. So it would be nice to store all the entries in the same array in memory. How does one indicate the bucket structure?

Linked List

A linked list consists of a group of things called nodes. Each node contains data and one or more pointers to other nodes. (A binary tree is a linked list.) This structure is used to solve our problem as follows:

Each node contains a symbol table entry and the sixteen bit address of the next node in the same bucket. There is also a 128 byte array containing the sixteen bit addresses of the first node in each bucket. An address such that the high byte is zero indicates that there are no more nodes in that bucket. The

```
LEXCMP:  LDAX  B        ;Load A with character addressed by BC.
         CMP   M        ;Compare with character addressed by HL.
         RNZ             ;If not equal, return with zero flag clear.
         INX   B        ;Advance to next character in each string.
         INX   H
         ORA   A        ;If not last character in both strings,
         JP    LEXCMP   ;Then continue comparison, else:
         XRA   A        ;Set zero flag and clear carry flag.
         RET             ;Return.
```

Listing 1: Subroutine LEXCMP [LEXical CoMPare] compares two ASCII strings. The addresses of the beginning of the strings are stored in the HL and BC registers. The last byte of each string is marked by its sign being set. The strings can be of any length, and don't have to be the same length. Equality of strings is indicated by returning with the zero flag set, otherwise the zero flag is clear.

```
BUCKET:  [block of 128 bytes, initially zero.]

HASH:    XRA   A        ;Clear A.
         XRA   M        ;XOR next character in string.
         INX   H        ;Advance to next character.
         JP    HASH+1   ;If not last character, continue hashing.
         ANI   3F       ;Use low 6 bits of result as hash code.
         MVI   H,00     ;Load HL with hash code.
         MOV   L,A
         DAD   H        ;Double hash code since addresses are two
                        ;bytes long.
         LXI   D,BUCKET ;Load DE with address of bucket pointer
                        ;array.
         DAD   D        ;HL Contains address of the pair of bytes
                        ;containing the address of the first node
                        ;in the bucket.
         RET             ;Return.
```

Listing 2: Subroutine HASH computes the hash code of the symbol "in" the HL registers by exclusive OR'ing the characters in the string.
HASH returns the hash code in A, and the address of the address of the first node in the bucket in HL.

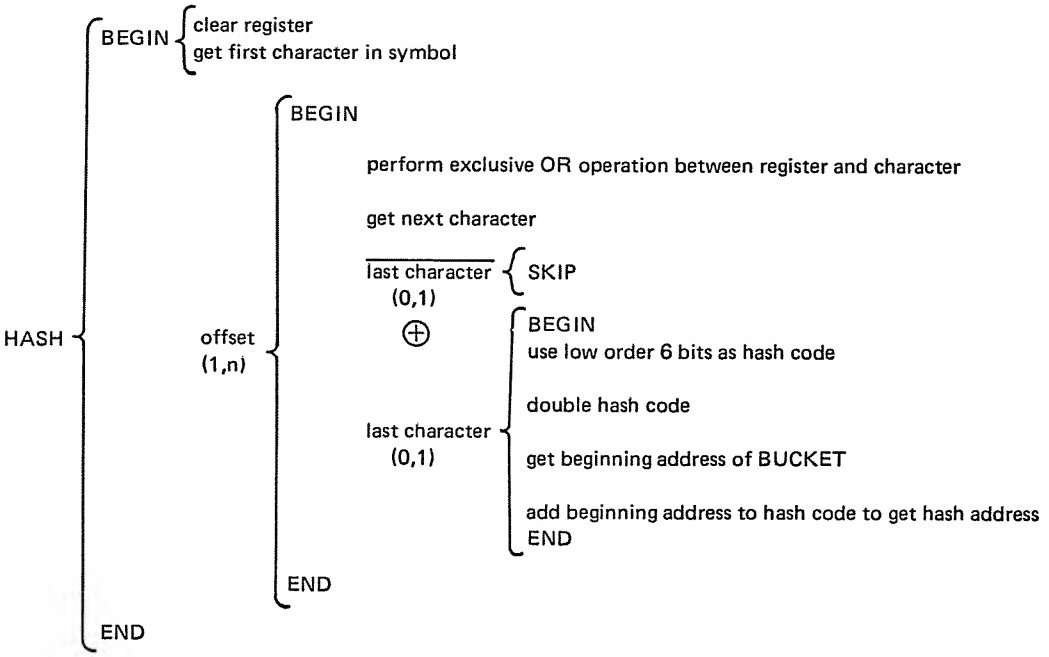


Figure 2: HASH computes the hash code of the symbol by exclusive-OR'ing the characters in the string. HASH returns the hash code and the address of the address of the first node in the bucket.

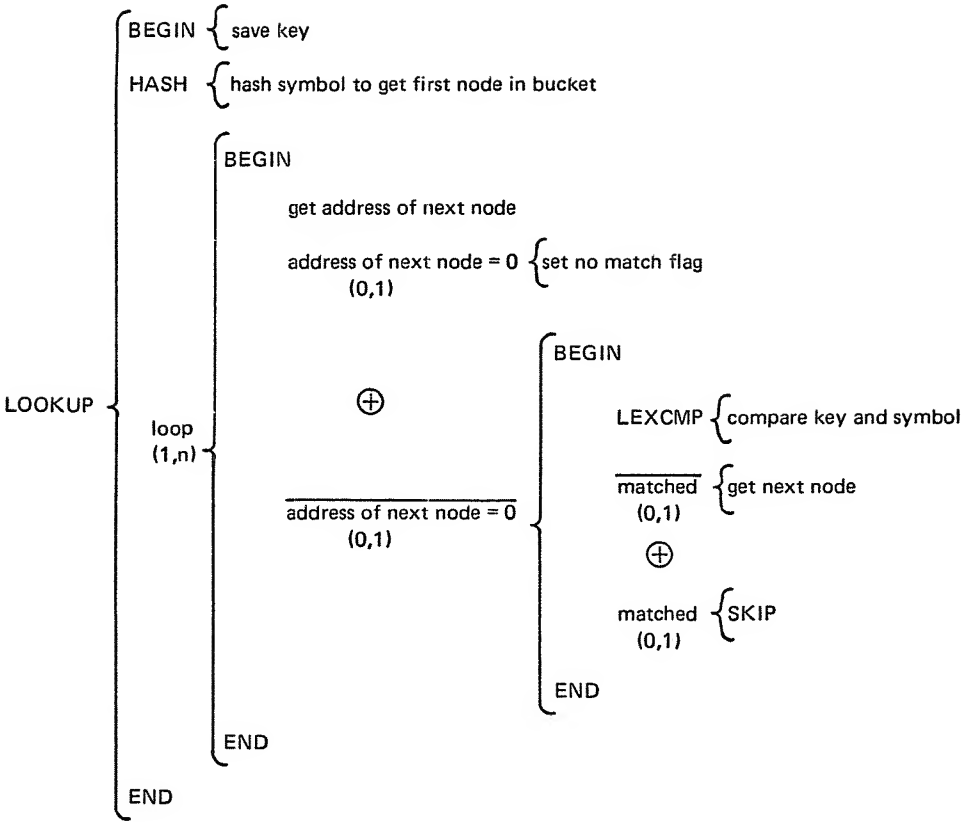


Figure 3: LOOKUP searches for the key symbol in the symbol table. When the symbol is matched, the address of the parameters represented by the symbol are returned and the match flag is cleared. If the symbol cannot be found, LOOKUP returns with the match flag set.

```
LOOKUP:  MOV   B,H      ;Save address of key in BC.
         MOV   C,L
         CALL  HASH     ;hash symbol: HL contains address of the
                        ;address of the first node in the bucket.
LOOP:    MOV   E,M      ;Load DE with address of next node in list.
         INX   H
         MOV   D,M
         MOV   A,D      ;Load A with high byte of address
         ORA   E        ;If DE is zero (symbol not matched)
         STC            ;then (set carry.
         RZ             ;      Return.)
         MOV   H,D      ;Load HL with address of node.
         MOV   L,E
         INX   H        ;Span (pass over) 2 byte field containing
         INX   H        ;address of next node in list.
         PUSH  B        ;Save address of key.
         CALL  LEXCMP   ;Compare key to symbol in node.
         POP   B        ;Restore address of key into BC.
         RZ             ;Return if successful match. (Carry cleared
                        ;by LEXCMP)
         XCHG           ;Load HL with address of address of next
                        ;node in list.
         JMP   LOOP     ;Continue search through bucket.
```

Listing 3: Subroutine LOOKUP searches for the symbol "in" the HL registers (the key) in the symbol table. On the symbol's first match, the address of the parameters represented by the symbol are returned in HL with carry clear. If the symbol cannot be found, LOOKUP returns with carry set.

logical structure of memory (as the program sees it) is different from the physical structure of memory. The buckets are stored in the same region of memory and there is no "crosstalk" between buckets.

Symbols consist of a variable length array of bytes containing 7 bit ASCII characters. The last character in the symbol is indicated by the sign bit being set, whereas the other characters have the sign bit clear. This is necessary so that both ends of the symbol are known.

A complete node consists of:

- 16 bit address of the next node byte 1 contains the low 8 bits, byte 2 contains the high 8 bits
- n bytes of symbol in ASCII.
- m bytes of parameters represented by the symbol.

We are now ready to look at some code.

The Routines

Subroutine LEXCMP (figure 1) is used to compare two ASCII character strings. The strings need not be of equal length.

LEXCMP works as follows: the first two characters are compared. If they are not equal, LEXCMP returns with a not equal flag set. If the first two characters are equal,

LEXCMP checks if those were the last characters in the strings. If so, LEXCMP returns with the not equal flag set; otherwise, the next two characters are compared, and so on.

Subroutine HASH (figure 2) computes the hash code of a symbol. HASH then computes the address of the pointer to the correct bucket.

HASH uses an exclusive-OR function to hash the characters. This makes it very easy to detect the end of the symbol, as only the last character will set the sign bit in the A register. BUCKET is the starting address of a 128 byte array containing the addresses of the first node in each bucket.

Subroutine LOOKUP (figure 3) searches for a key. If the key cannot be found in the table, LOOKUP returns with a not found flag set; otherwise, LOOKUP returns the address of the parameters associated with the symbol, and clears the not found flag.

Subroutine INSERT, shown in figure 4, inserts a node into the symbol table. This is very easy to do using the linked structure. Only two addresses must be moved. The first node in the bucket is linked to the new node, and the address in the BUCKET array links to the new node; thus, the new node becomes the first node in the bucket.

Subroutine DELETE, figure 5, removes a node from the symbol table. DELETE requires that the node to be deleted was the last node inserted into the bucket. This is not as severe a limitation as one might think. (In a compiler such as ALGOL, symbols go out of existence in reverse of the order they came into existence; exactly what goes on here.) This limitation simplifies the DELETE operation considerably, and also simplifies reclamation of space (reusing memory freed by deleting nodes.)

DELETE moves the address in the link field of the first node in the bucket into the proper element of the BUCKET array. Thus the second node in the bucket becomes the first node. Notice that the INSERT and DELETE operations are exactly PUSH and POP operations, where the stack is organized as a linked list instead of a contiguous array.

This set of routines could be the basis for any symbolic data handling program. The structures are not limited to ASCII and could be used for any code, for example, a phonetic language. This system is the symbolic backbone of a compiler or interpreter: LISP could be kept very happy. Notice that you can have a symbol defined many times: The most recent assignment is valid (it has "extent"), yet the older ones still exist (they have "scope"). This is caused by this insertion and deletion method, and is the basis for block structured languages.

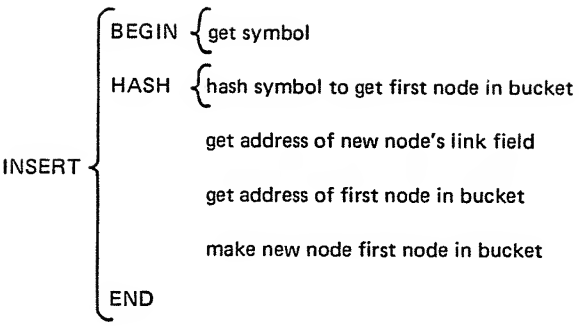


Figure 4: INSERT inserts a node into the symbol table. The new node becomes the first node in the bucket.

```
INSERT: PUSH  H       ;save address of new node.
        INX   H       ;break (move up to) to symbol.
        INX   H       ;span link bytes.
        CALL  HASH    ;hash symbol: HL contains address of address
                      ;of first node in bucket.
        POP   D       ;Load D with address of new node's link field.
        MOV   A,M     ;Get address of first node in bucket.
        STAX  D       ;Set link of new node to point to first node in
                      ;bucket.
        MOV   M,E     ;Make new node first node in bucket.
        INX   H       ;Do above to second bytes of addresses.
        MOV   A,M
        MOV   M,D
        INX   D
        STAX  D
        RET           ;Return.
```

Listing 4: Subroutine INSERT inserts the node addressed by HL into the symbol table. The new node becomes the first node in the bucket.

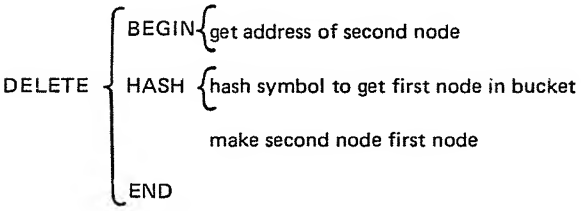


Figure 5: DELETE removes a node from the symbol table. This node must be the first node in the bucket. The second node in the bucket becomes the first node.

```
DELETE: MOV   C,M     ;Load BC with address of second node.
        INX   H
        MOV   B,M
        INX   H
        CALL  HASH    ;Hash symbol: HL contains address of address
                      ;of first node in bucket.
        MOV   M,C     ;Make second node in bucket the first node in
                      ;bucket.
        INX   H
        MOV   M,B
        RET           ;Return.
```

Listing 5: Subroutine DELETE removes the node whose address is in HL from the symbol table. This node must be the first node in the bucket. The second node in the bucket becomes the first node.

Table 1

Address	Initial Data	After Insert BAT	After Insert CAT	After Insert ACT	After Insert TAB	After Delete ACT	After Insert TAC
0000	0000	0000	0000	0000	0000	0000	0000
0002	0000	0000	0000	0000	0000	0000	0000
....							
002C	0000	0000	1008	1010	1010	1008	1020
002E	0000	1000	1000	1000	1018	1018	1018
....							
007E	0000	0000	0000	0000	0000	0000	0000
....							
1000	0000	0000	0000	0000	0000	0000	0000
1002	'B,'A	'B,'A	'B,'A	'B,'A	'B,'A	'B,'A	'B,'A
1004	'T,00	'T,00	'T,00	'T,00	'T,00	'T,00	'T,00
1006	1234	1234	1234	1234	1234	1234	1234
1008	0000	0000	0000	0000	0000	0000	0000
100A	'C,'A	'C,'A	'C',A	'C',A	'C',A	'C',A	'C',A
100C	'T,00	'T,00	'T,00	'T,00	'T,00	'T,00	'T,00
100E	4688	4688	4688	4688	4688	4688	4688
1010	0000	0000	0000	1008	1008	1008	1008
1012	'A,'C	'A,'C	'A,'C	'A,'C	'A,'C	'A,'C	'A,'C
1014	'T,00	'T,00	'T,00	'T,00	'T,00	'T,00	'T,00
1016	AF00	AF00	AF00	AF00	AF00	AF00	AF00
1018	0000	0000	0000	0000	1000	1000	1000
101A	'T,'A	'T,'A	'T,'A	'T,'A	'T,'A	'T,'A	'T,'A
101C	'B,00	'B,00	'B,00	'B,00	'B,00	'B,00	'B,00
101E	6878	6878	6878	6878	6878	6878	6878
1020	0000	0000	0000	0000	0000	0000	1008
1022	'T,'A	'T,'A	'T,'A	'T,'A	'T,'A	'T,'A	'T,'A
1024	'C,00	'C,00	'C,00	'C,00	'C,00	'C,00	'C,00
1026	7897	7897	7897	7897	7897	7897	7897

This is one method of storing a hashed symbol table. Assume that the symbols we will be working with are already in memory starting at hexadecimal address 1000. Each entry consists of three parts:

- *the address of the symbol which follows this one in the bucket;*
- *the symbol itself;*
- *the value represented by the symbol.*

For the purposes of this discussion, the symbols are assumed to be four characters long and followed by a two byte value.

Column one in table 1 shows how memory is initially arranged. BUCKET occupies memory from hexadecimal address 0000 to 007E. This allows arrangement of 64 buckets. All pointers in the symbol and hash tables are initialized to zero. A pointer value of zero indicates that this symbol is the last one in that particular bucket since that address is not defined in the symbol table.

Insertion

The first symbol that will be entered into the BUCKET table is BAT. The symbol BAT hashes to hexadecimal location 002E. The pointer at address 002E is 0000. This means that there are no symbols in this particular bucket. We now set this location equal to hexadecimal 1000 which is the address of the symbol.

The next symbol we wish to insert is CAT. This symbol hashes to hexadecimal location 002C. Since this location is also equal to zero, indicating no symbols in the bucket, we point it to CAT at location 1008.

The third symbol to be inserted (ACT) hashes to the same location that CAT did since it contains the same letters. Since there are already symbols in this bucket, we search the entire bucket to make sure that this symbol is not already contained within the bucket. Since it is not, we will place it at the head of the bucket. This is done by having the first pointer in the bucket point to this symbol (hexadecimal address 1010). The pointer that ACT has is adjusted to hexadecimal address 1008 to point to CAT. CAT's pointer is still 0000 indicating that it is the last symbol in the bucket.

Deletion

The particular format that we have adopted requires that any symbol that is to be deleted must be the first node of a bucket. This implies that it was the last symbol added to that bucket.

Suppose we want to delete the symbol ACT. ACT is the last symbol that was inserted into the bucket located at hexadecimal address 002C. If the pointer at this location is changed to point at the second symbol in the bucket, ACT is effectively eliminated from the hash table.

If this method is employed, the minimum number of pointers need to be changed when inserting or deleting a symbol. Insertion requires that the pointer at the head of the bucket point to the new symbol location, and the new symbol points to the node that used to be at the head of the bucket. Deletion requires changing only the pointer at the head of the bucket from the first node to the second node of the bucket. ■

TABLE:	00	N	O	P
+4:	01	L	D	R
+8:	02	S	T	R
+12:	03	A	D	D
+16:	04	A	N	D
+20:	05	R	O	T
+24:	06	C	L	R
+28:	07	S	T	P

Figure 1: An Opcode Table Organized for Direct Access. Note that with this particular organization the first data byte of each entry is related to the address of the entry within the table, in a sorted sequence.

Making Hash With Tables

Terry Dollhoff

Hashing is a technique used to speed up table searching operations by making position in the table depend upon the data. Many newcomers to programming reject hashing as an overly complicated technique useful only by the designer of exotic systems software, but this is not the case. Any large program, written for fun or profit, may include tasks of accessing, storing, or modifying entries in a table or array. Most game playing programs include a number of such tasks. Application of hashing techniques can often dramatically improve the performance of these programs. This article will explore the use of hashing (sometimes called key-to-address transformation) as a simple but effective mechanism for accessing stored data. These techniques can be used in applications where the data is organized randomly and where each item has a unique key associated with it. For example, consider a table that contains computer opcode mnemonics and their associated value as used in an assembler; by using the opcode value as a key this table could be used to determine the mnemonic associated with any particular value. Such a table is an integral part of any disassembler.

In any computer, a particular entry in a table can be specified by the starting address of the entry. Locating an item in a table implies that the starting address for that item must be determined. One possible method that can be used to determine the address, and by far the most common method, is to examine each item sequentially, starting with the first item, until the desired item is located and hence the item address determined. This approach is termed a linear search and as you can see by the the 8080 subroutine of listing 1, it is simple to code. The big disadvantage of a linear search is that it is costly in terms of processing time because, on the average, at least one half of the table entries must be examined before locating the desired item. If the table is moderately large and numerous accesses are required, then the table lookup processing time will constitute a significant part of the total processing time.

An alternative to the linear search involves storing the information in a sorted fashion based upon the key. However, even the best known algorithms for locating data in a sorted table require an average of Log_2N tests, where N is the table size. Therefore, a table with, let's say, 500 entries requires an

Listing 1: Typical 8080 code sequence for a linear search of a table until the first byte of the current table entry matches the value in the accumulator. In this listing, the HL register pair must be preset to the address of the table, the DE register pair must be set with the number of bytes per table entry, the B register must contain the number of entries to search (maximum 255) and the key value sought must be loaded in A. This is by no means the only possible 8080 linear search strategy.

```
FIND:   CMP   M      Check for a match;
        RZ           If so then exit;
        DAD   D      Advance to next table entry;
        DCR   B      Decrement count;
        JNZ   FIND   Continue till end;
        JMP   ERR    Table exhausted, treat as error;
```

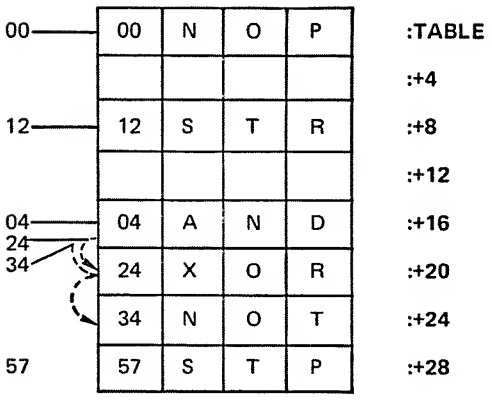


Figure 2: A Hash Accessed Table. Note that with the hash algorithm described in the text, three elements of this table map into identical starting entries, resulting in a rehash requirement indicated by the arrows and dotted lines.

average of nine tests to locate an arbitrary item. Although this is a considerable improvement over the linear search, which would require an average of 250 tests to locate an item, hashing techniques require considerably fewer tests than either method, without the added burden of sorting.

The Key

The fundamental idea behind any hashing technique is that instead of searching the table to determine the address of a particular entry, an attempt is made to calculate the address using the key. That is, a subroutine is written which, when given any desired key, calculates the table location containing the item associated with that key. If this calculation is successful, then the desired item is located with a single search.

The first step is to determine the key. This choice will depend upon the intended use of the table. In the opcode table mentioned earlier, the opcode value is the key since all lookup requests are of the form: "What is the mnemonic for the opcode X?" On the other hand, if this same table were incorporated in an assembler or compiler, then the mnemonic would be the key because requests are now of the form: "What is the opcode value for mnemonic X?". In all of our examples, we will assume that the opcode value is the key.

Direct Access Hash

Imagine that there are only a limited number of opcode values and it so happens that, although the value is eight bits long, the opcode is uniquely determined by the rightmost three bits. If a table, called TABLE, is created with eight 4 byte entries, and the mnemonic and value for each opcode is placed in the table entry whose address is found by multiplying the rightmost three bits of the opcode by four and adding the results to the base address of the table, then a simple subroutine can calculate the precise location of any entry. That subroutine, shown in listing 2 for an 8080, simply strips off the rightmost three bits of the key, multiplies them by four, and adds in the starting address of the table as shown in figure 1. Entries are added to the table in the same manner. Tables of this type are called *direct access* and are most commonly used for conversions; that is, converting from one character code to another, from opcode values to mnemonics, etc. In many direct access tables the actual key is not even stored in the table since a comparison is not necessary to determine the proper entry.

Open Hash

The direct access method would obviously break down if certain opcode mnemonics were associated with values whose rightmost three bits were equal. In this case, where direct access is infeasible, the algorithm must be slightly modified. A subroutine is still used to calculate the address, but since it is no longer possible to

Editor's Note:

In this article, we represent several algorithms in a structured pseudo code form appropriate to the discussion. These algorithms are referenced by numbers in brackets, as in [n] for algorithm n. Each algorithm should be thought of as a formal procedure, which in practice would be called as a subroutine.

*Listing 2: Typical 8080 code sequence for direct hash with a table of eight entries, each entry being four bytes in length. In a direct hash approach, the actual data value (in this case, a number from 0 to 7) being sought is used to determine the offset in the table directly. Here the calculation is made according to the formula: ADDR := BASE + 4 * (A & 7) where A is the value of the entry being sought, BASE is the starting address of the table, and ADDR is the effective address of the table element involved.*

```
FIND:   LXI     H,TABLE     HL:=Table pointer;
        ANI     7           Extract rightmost three bits;
        RLC                 Multiply by four;
        RLC
        ADD     L           Add the table address;
        MOV     L,A
        MVI     A,0
        ADC     H
        MOV     H,A         HL:=Entry address;
        RET
```

successfully calculate the location of all entries, some type of searching algorithm must be employed to pinpoint the position of the entry, given the calculated position. The initial predicted position of the table item is called the *hash index* and the procedure which produces the hash index is called the *hashing function*. For the remainder of our discussion, HASH is used to denote that subroutine and therefore HASH(K) denotes the hash index for a particular key, K.

Before considering how the information is initially entered in a hash table, it may be useful to examine the process used to locate an arbitrary entry in a hash accessed table. If KEY is used to denote the key associated with the desired entry, and TABLE, a table consisting of N entries (each of which are B bytes long), then the algorithm to locate the entry that is associated with KEY, using hash techniques, is as follows:

```
[1]   1.  I,J := HASH(KEY);
      2.  do until (I=J-1) [worst case end test for search failure];
      3.     if @(TABLE + I * B) = 0 then [element not present, search failure];
      4.        do; call ERROR; return; end;
      5.     if @(TABLE + I * B) = KEY then
      6.        return [the item has been located];
      7.     I := I + 1;
      8.     if I = N then I := 0 [wrap around table space limit];
      9.  end;
     10.  call ERROR [element not present, search failure];
```

In this algorithm, specified in a structured pseudo code form, step 1 calculates an initial estimate of the location of the item associated with KEY, the hash index. This value is saved in J for the worst case end test in the *do until* construct of step 2. In steps 3 and 4, the algorithm tests for a null entry end of search criterion and calls an ERROR routine if this is detected. Return to the calling program follows detection and flagging of the search failure condition. Then the algorithm tests to see if the current entry is equal to KEY at step 5; if this condition is found, the algorithm terminates with a return operation at step 6. Otherwise, the next index is calculated at step 7, an end around wrap condition is tested at step 8, and the do loop is closed at step 9 with an end statement. If the loop execution ends through the test on line 2, step 10 is reached and an error condition is flagged before an automatic return assumed after the last line of such a procedure.

Consider again the opcode table example. If the hash procedure is defined as HASH(K) = REMAINDER(K/8), then each table item shown in figure 2 can be located by at most three searches using algorithm [1].

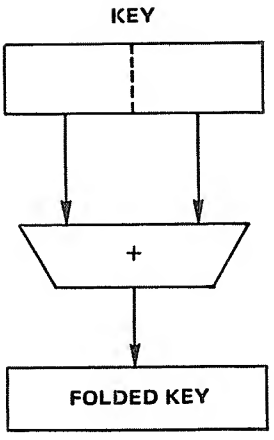


Figure 3: Folding Keys. When it is desired to retain the significance of all the bits in a key while compressing the total number of bits used, folding by some operation such as addition can be used.

Defining HASH

In choosing a hash function, you must attempt to define a general procedure, using a minimal number of simple computations, which produces an even distribution of hash indices for a random selection of possible keys. If we knew that all op codes were even numbers, then the hashing function HASH(K) = REMAINDER(K/8) would not be efficient, because it will produce only even numbers. This simple example illustrates that the hashing function must be carefully selected to suit the particular application. It should also be noted that it is not necessary for the key to be a numeric value. If alphanumeric or other keys are used, the hashing function should ignore the data type and simply perform numeric or logical manipulations of the key as though it were numeric.

One of the most widely utilized, and historically the first, hashing function has already been mentioned. If N is the size of the table (in terms of the number of entries, not the number of bytes) the hash index is the remainder of the key divided by N. More precisely stated, HASH(K) = REMAINDER(K/N). In a machine such as the 8080 which lacks division capability, this function will be made significantly faster by restricting the length of the table to a power of two (ie: $N = 2^M$). If $N = 2^M$, then the REMAINDER (K/N) also happens to be the rightmost M bits of K and a divide operation is no longer required. The remainder is selected by a logical AND operation.

The remaindering function will not produce well distributed hash indexes if many of the entries end with the same bit sequence. This situation is frequently encountered when dealing with alphanumeric data. Changing the table size to a prime number usually improves distribution, but now we are back to the unwanted divide operation for calculating the remainder. There are two other alternatives to this problem. The first is a technique called *folding* as diagrammed in figure 3. This method applies the remaindering algorithm to the bit string that is obtained by adding the upper half of the internal binary representation of the key to the lower half. This minor improvement minimizes the effect of patterns that may occur within the key. You should be careful what improvisations are made to the folding technique. For example, substituting a logical AND for the add sounds good, but will merely make matters worse. If in doubt, try experimenting with various keys by examining the effects of key value in a test program to grind out hash indices.

A second method for minimizing the effect of similar bit patterns in the key, best applied to tables of size 2^M, is called *squaring*. This consists of selecting the center M bits of the number that is obtained by multiplying the key by itself. Since the middle bits of the product depend upon all of the bits in the key, this method generally produces a uniform distribution of hash indices.

Since the squaring method is safest, it may appear that one should always use it. This is certainly not the case because the purpose of hashing is to save processing time and although squaring is the most general technique, it is unfortunately the slowest since it relies on a multiply operation which the 8080 and many other small processors lack. It is often acceptable to settle on a slightly less efficient hash function if such a function is substantially faster. The guideline for selecting the hash function is to employ a more complex function only in those specific cases where a simple function fails to produce an adequate distribution of hash indices. But remember, any hash function is better than a linear search. Why? A linear search is a hash access where HASH(K)=0 for all values of K, therefore any distribution is better than none. This degeneracy is evident in algorithm [1] when the data item sought is not in the table, and the algorithm searches every location.

Multibyte Hash

Until now, we have tacitly assumed that the entire key can be contained in one byte. This is impractical, and the hashing concept is easily extended to cover those cases where the key occupies more than a single byte. If the key is continued in byte locations (K_1, ... K_j) then a multibyte hash function, HASHM, can be defined in terms of any of the previous hash functions as HASHM(K,J) = HASH(K_1+ ... +K_j). That is, any of the single byte hash functions are applied to the sum, ignore carry, of the bytes in the multibyte key. As you see in figure 4, this is similar to the folding technique just mentioned.

Another possibility for a multibyte hash, which should be used with some degree of caution since it may not provide an even distribution, is to apply a single byte hash function to the last byte (or any other byte of your choosing) of the multibyte key. This eliminates the time required to add the words of a multibyte key. As usual, the programmer is faced with a time versus efficiency tradeoff.

Guidelines

In summary, the sole purpose of a hashing function is to calculate an initial table index for a linear search, given a specific key. There is no one best algorithm and the number of algorithms available is bounded only by your imagination. The general guidelines to follow when designing your hashing function are:

1. Keep it simple – Remember, the goal is to locate an item in the minimum amount of time. If the perfect hash requires more time than a linear search, it is useless!
2. Insure an even distribution; beware of weird bit patterns in the key.
3. Check out the operation of the function prior to employing it as a hash function. There is often an overwhelming urge to give it the smoke test, but hash indices are used to form memory addresses so it may be difficult to isolate bugs in the hash function after you've incorporated it into a table lookup procedure. Save yourself some time, check the table lookup subroutines first.

Building the Table

Obviously, for the hash access algorithm to operate smoothly, the table items must have been entered into the table properly. The relative ease with which entries can be made in a hashed table is an important advantage of hash techniques. Remember, even though a sorted search is reasonably efficient for locating an entry, the entire table must be sorted before any access is allowed. Thus, if accesses were to be intermixed with entries, the algorithm would be grossly inefficient due to the amount of resorting required.

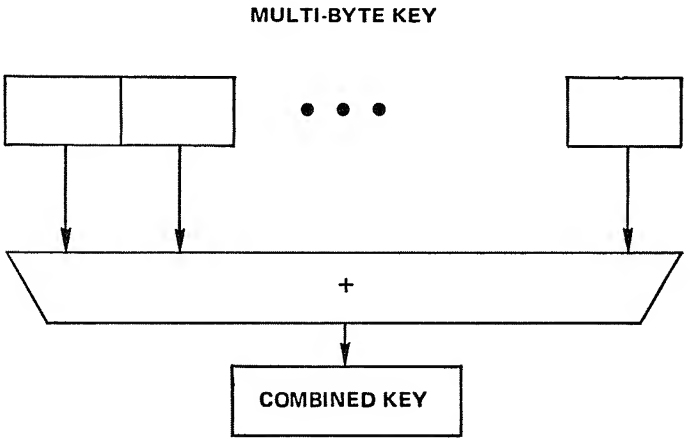


Figure 4: The principle of folding key elements can be extended to a multibyte key. The multibyte hashing scheme might be employed where a key is a character string field.

Before any entries can be made in the hash table, the key field of the table must be initialized to some flag value which is not encountered as a possible key. If a table entry contains this value, then it can be assumed that the entry is unoccupied. The most common value used to designate an empty table entry is the integer zero, and assuming this to be the case, the algorithm to add an item associated with KEY, to the table of N entries (each B bytes long) is:

```
[2]   1.  I,J := HASH(KEY);
      2.  do until (I=J-1) [worst case end test for search failure];
      3.     if @(TABLE + I * B) = 0 then
      4.        do;
      5.           [enter the item at (TABLE + I * B)];
      6.           return;
      7.        end;
      8.     I := I + 1;
      9.     if I = N then I := 0  [wrap around table space limit];
     10.  end;
     11.  call ERROR [no room left in table];
```

Notice that the lookup algorithm [1] and the entry algorithm [2] are very similar in nature. The loop control is identical, and the only difference is in the actions taken. It is quite possible to make an automatic entry occur whenever a key is not found as indicated by a null key value found during a search. The following algorithm combines both operations.

```
[3]   1.  I,J := HASH(KEY);
      2.  do until (I=J-1) [worst case end test for search failure];
      3.     if @(TABLE + I * B) = KEY then
      4.        return [the item has been located];
      5.     if @(TABLE + I * B) = 0 then
      6.        do;
      7.           [enter the item at (TABLE + I * B)];
      8.           return;
      9.        end;
     10.     I := I + 1;
     11.     if I = N then I := 0 [end wrap around table space limit];
     12.  end;
     13.  call ERROR [if this point is reached, table is full];
```

In addition to adding or locating entries, it may also be necessary to delete entries. To delete an item, you might think that we could merely locate the item and then set the table entry to zero, thus making it available for future entries. However, if that approach were taken, not only would the desired entry be deleted, but other entries might be made inaccessible. The reason that other entries would be lost is that the searching terminates when an unused location is found. As an example, setting the entry at (TABLE + 20) in figure 2 to zero would also make the entry at (TABLE + 24)

TABLE:	00
+1:	N
+2:	O
+3:	P
+4:	01
+5:	L
+6:	D
+7:	R
+8:	02
+9:	S
+10:	T
+11:	R
.	.
.	.
.	.
+28:	07
+29:	S
+30:	T
+31:	P

Figure 5: Horizontal Organization of Tables. In this method of organization, all the bytes of a data entry are assigned to contiguous addresses.

% Items Occupied	Linear Search	Sorted Search	Hash Access Method With Rehash Indicated Below			
			Linear	Random	Day's	Wt Inc
10%	25	5.6	1.1	1.1	1.1	1.0
50%	125	7.9	1.5	1.4	1.4	1.4
75%	187	8.5	2.5	2.0	2.0	1.9
90%	225	8.8	5.5	2.8	2.7	2.5

Table 1: Comparison of Table Access Methods. This table gives the results of an experiment with random data to test out the various methods of access. The tables were filled to the percentage levels indicated at the left. A table size of 500 possible entries was used. The access methods shown are described in text.

inaccessible. Therefore, an alternate scheme must be used to delete entries.

The first step is to select a deleted entry flag that is distinguishable from the unused entry flag and is also not allowable as a key. Then, whenever an entry is to be deleted this new value replaces the entry. The new flag indicates that the entry is available for future additions to the table but does not terminate a search operation. If 0 is used to denote an unused entry and −1 is used to denote a deleted entry, then the complete hashing algorithm is:

```
[4]   1.  I,J := HASH(KEY);
      2.  do until (I=J−1) [worst case end test for search failure];
      3.     if @(TABLE + I * B) = KEY then
      4.        do;
      5.           if [entry is to be deleted] then [delete the entry]
      6.              @(TABLE + I * B) := −1;
      7.           return [item has been located ];
      8.        end;
      9.     if @(TABLE + I * B) = 0 then [this is a null entry so]
     10.        do;
     11.           [enter the item at (TABLE + B * I)];
     12.           return;
     13.        end;
     14.     I := I + 1;
     15.     if I = N then I = 0 [end wrap around table space limit];
     16.  end;
     17.  call ERROR [if this point is reached, table is full];
```

This algorithm either locates an item or adds the item to the first available location. If an item is to be deleted it is first located and then the key field of the table entry is set to −1.

TABLE:	00	BYTE0
+1:	01	+1
+2:	02	+2
+3:	03	+3
+4:	04	+4
+5:	05	+5
+6:	06	+6
+7:	07	+7
+8:	N	BYTE1
+9:	L	+1
+10:	S	+2
+11:	A	+3
+12:	A	+4
+13:	R	+5
+14:	C	+6
+15:	S	+7
.	.	
.	.	
.	.	
+24:	P	BYTE3
+25:	R	+1
+26:	R	+2
+27:	D	+3
+28:	D	+4
+29:	T	+5
+30:	R	+6
+31:	P	+7

Figure 6: Vertical Organization of Tables. In this method of organization, a multibyte table element is treated as "n" single byte subtables where "n" is the number of bytes in each entry. Each of the "n" subtables has a length (in bytes) equal to the number of elements in the table.

Collisions

A *collision* occurs whenever HASH(KEY1) = HASH(KEY2), but KEY1 ≠ KEY2. As discussed earlier, a good hashing function will avoid this condition, but the problems caused by collisions cannot be ignored. Note for example that the hash index for opcodes 04, 24 and 34 in the table shown in figure 2 is 4 and hence these entries collide.

What happens when two entries collide? The only solution we've discussed thus far is to search the table, in a circular fashion, from the point of impact as in algorithms [1] to [4]. If, in general, a collision occurs, then the resulting search, good or bad, is called a *rehash*. The process mentioned above, namely, searching the table in a circular fashion from point of impact, is called a *linear rehash*, and as you might expect falls into the bad category. Other more efficient algorithms will be discussed later.

If we denote the rehashing algorithm by REHASH, then the general hashing lookup algorithm may be restated in its final form:

```
[5]   1.  I,J := HASH(KEY);
      2.  K := 0;
      3.  do until (REHASH(I,J)=J) [worst case end test for search failure];
      4.     if @(TABLE + I * B) = KEY then [we have a match so]
      5.        do;
      6.           if [entry is to be deleted] then [delete the entry]
      7.              @(TABLE + I * B) := −1;
      8.           return;
      9.        end;
     10.     if ((K=0) & [deletion or null element @(TABLE + I * B)]) then
     11.        K := I [save last available table entry index];
     12.     if @(TABLE + I * B) = 0 then [this is a null entry so]
     13.        do;
     14.           [enter the item at (TABLE + B * K), next available slot];
     15.           return;
     16.        end;
     17.     I := REHASH (I,J) [REHASH results in 0 ≤ I ≤ N where N is table size];
     18.  end;
     19.  call ERROR [if this point is reached, table is full];
```

The linear rehash that we've been using implicitly in [4] as steps 14 and 15 is described as REHASH(I) = (I+1)[mod N], where (I+1)[mod N] means that if (I+1) is greater than or equal to N, then N is subtracted from the value (I+1). This insures that the table is searched in a circular manner. The operation X[mod N], called X modulo N, is used in most rehashing algorithms to limit the range. Mathematically, it is the remainder of X/N; but whenever we

use X[mod N] it can be calculated as described above (ie: subtract N if X is greater than or equal to N). Here again we have avoided the use of a divide operation to provide a more efficient function. Note that step 10 includes a check which reclaims deleted entries, a process not included in algorithm [4].

Improved Rehash

The problem with the simple linear rehash is that the table will not fill uniformly. This condition is referred to as *clustering* and causes an increase in the average number of tests required to locate an item in the table. As an example, a cluster can be seen forming at TABLE+16, +20, and +24 in the table shown in figure 2.

There are a number of nonlinear algorithms which perform the rehash function without causing the clustering problems mentioned above. Although the computer science literature abounds with such algorithms, a majority of them fall into one of three classes. An attempt has been made to select the simplest and best from each class and present them here.

Pseudorandom Rehash

The first class of rehashing algorithms is the *pseudorandom* rehash and is based upon a pseudorandom number generator. The pseudorandom number generator used is not important, but it must be of the non-repeating variety. That is, it must generate all possible values before any previous value is repeated. It must also generate all of the integers in the range 0, ..., N where N is the table size. The following simple procedure incorporates a common random number generator and will perform the rehash function for any table of size $N = 2^M$. The variable R is internal to the rehashing function, but it must be preset to one whenever the function HASH is initiated (ie: step 1 of algorithm [5]).

```
[6]  REHASH (I,J):
       1.   R := REMAINDER (R*5 / N*4);
       2.   REHASH := (R/4 + J) [mod N];
```

If you're seeking the most efficient implementation of this one, the REMAINDER(R*5/N*4) is just the rightmost M+2 bits of R*5 because $N=2^M$ and $4*N=2^2*2^M=2^{M+2}$. Furthermore, the divide operation in step 2 can be replaced by a right shift of two positions. Finally, if you think of R*5 as R*4+R, then it's easy to see how to reduce that multiply operation to left shift and addition operations.

Let's look at the sequence generated by this rehash routine. If our table is eight entries long and the initial hash index is, let's say, 4, then R takes on the values 1,6,7,4,5,2,3,1, so the table would be searched in the order 4(initial index), 5,2,3,0,1,6,7. How does this avoid the clustering situation? If we chose another initial index, say, 5, then the table is searched in the order 5(initial index),6,3,4,1,2,7,0. As you see, the entry searched after entry 5 will depend upon the initial index. If the initial index was 4, then 2 is searched after 5; but if the initial index was 5, then 6 follows 5. In a linear search, 6 always follows 5. This dependence upon the initial index is what avoids the clustering.

Quadratic Rehash

A second class of algorithms for rehashing is the *quadratic* rehash and these are based upon a quadratic function. The major drawback with most algorithms in this class is that they search only one half of the table, so two different rehashing algorithms are required. The most efficient quadratic rehash, and one which does search the entire table, was first introduced by Colin Day *[see bibliography, reference 1]*. Day's algorithm can only be applied to a table whose size is a prime number that produces a remainder of 1 when it is divided by 4 (eg: 5=4*1+1, 401=4*100+1). At first glance, this appears to place a great many restrictions on the allowable size of the table; but don't despair, because experience will show that a number satisfying the required condition can be found very near any desired value. Be certain that you use an acceptable number or the procedure will not search all locations of the table. Like the last rehashing function an internal variable is used. The variable, R, must be preset to (−N) whenever the function HASH is called. The quadratic rehash process is (remember that the mod operation is just a conditional subtraction):

```
[7]  REHASH(I,J):
       1.   R := R + 2;
       2.   REHASH := (I + |R|) [mod N];
```

If we look at the sequence generated by this procedure, we see that R takes on the values (for a table of size 11=4*2+3) −11,−9,−7,−5,−3,−1,1,3,5,7,9,11. Therefore, if the initial index were 4 the table would be searched in the order: 4(initial index), 2,9,3,6,7,8,0,5,1,10. One major difference between this algorithm and the random rehash is that this one calculates the next index based on the previous one. The random rehash calculates the next index based on the initial index.

Weighted Increment Rehash

The last, and probably the simplest, method for performing the rehash is called a *weighted increment [see bibliography, reference 2]*. This one is unique because it uses the hash index to calculate an increment which is in turn used to step through the table. The table size is again restricted to a power of 2, and whenever the function HASH is called, the variable R is preset to (2*J+1)[mod N], where J is the initial hash index. The weighted increment method is:

```
[8]   REHASH(I,J):
        1.    REHASH := (I+R) [mod N];
```

This process is very much like a linear rehash. In fact if R were always set to 1 it would be a linear rehash; however R depends on the hash index. If our table is eight entries long and the initial index is 5 then R=2*5+1[mod 8]=11−8=3 and the table items are searched in the order 5,0,3,6,1,4,7,2. Since the increment is a constant for any particular hash index, we can improve the basic hash algorithm when using this rehash technique. You will notice that all memory references are of the form (TABLE+I*B), where B is the number of bytes. We can avoid that multiply by including it in the computation of R. If we let R=((2*J+1)[mod N])*B, then all of the table references become (TABLE+I). If we also initialize I to TABLE+HASH(KEY) we can make all references as just (I).

Laying Doubts to Rest

You might conceivably ask, "What is gained by using a complex rehashing function?"; or if you're one of the more cynical observers, "Why use hashing at all?". In an attempt to answer these questions, a simple experiment was performed. First a table of approximately 500 entries was filled with randomly generated entries and then each entry was located in the table using the lookup technique under test. This simple experiment provides an insight into the comparative efficiency of table lookup algorithms. Table 1 summarizes the results of the experiment. This data clearly illustrates that there is significant improvement in table lookup time when hashing is utilized. Furthermore, when a complex rehashing algorithm is incorporated in the search procedure, the statistics are again improved. It is worth noting again that, although the number of tests for a sorted table is not tremendously large, the approach is very inefficient if the table must be accessed before being filled with entries.

One other surprising fact about the average search length (the number of tests required) for hash accessed tables is that it does not depend upon the length of the table. Rather, the search length depends only upon the load factor or the percentage of occupied items in the table. This means that you can expect the average search time for a table of size 10,000 to be about the same as the search time for a table of size 500! This is surely not the case with the linear or sorted search. While the average linear search length skyrockets to 4,500 (for a 90% full table of size 10,000), the average hash search length remains at less than six!

Although table 1 seems to indicate that the weighted increment is most efficient, we must be careful not to read too much into these results. The statistics in table 1 were obtained using randomly generated keys in the test program. When actual keys are used the search statistics will vary somewhat because actual keys are rarely perfectly random. For example, the search length for a weighted increment search is adversely effected by bit patterns in the key. The best way to insure that you are using the most optimal search procedure is to repeat the experiment with a sample of actual keys. If a finely tuned algorithm is not important, then the weighted increment is probably the better choice because it is simple and can be applied to any format of table. As we will see shortly, most of the algorithms work best if the table is rearranged in memory.

Application

There are a number of "tricks" which can be used to improve efficiency. A number of them have already been mentioned. Throughout our discussion we have assumed that each table entry occupies more than a single byte. If each table entry is B bytes long, then the typical memory reference is (TABLE+I*B). It would be desirable to eliminate or at least reduce the multiply operation. We already discussed how to eliminate the multiply if a weighted increment rehash is used. Another method to eliminate a multiply is table reorganization.

All of the tables discussed so far were horizontally organized. This means that the items are stored as shown in figure 5. This is the most common table organization. An alternative organization is a vertical organization such as in figure 6. If you have organized your table vertically then the first byte of an item is addressed by (TABLE+I) and the multiply is gone. All of the other bytes in the item are addressed by (BYTEN+I) where BYTEN is the address of the n^{th} byte of the first item. Thus by organizing the data vertically we eliminate a multiply operation. This vertical arrange-

ment is practical from other aspects also. Consider searching the table for all items containing a specific value in the third byte. Since the third byte of each item is stored sequentially this search operation is simplified.

Conclusion

We have tried to show that hashing is not nearly as complicated as you might have thought. By using these techniques perhaps you can regain a valuable slice of your microprocessor's processing load.■

GLOSSARY

Clustering: Grouping of elements within a table caused by equal hash indices.

Collision: Two elements with the same hash index.

Direct access hash: A hash algorithm which precludes collision. That is, no two elements have identical hash indices.

Disassembler: A program to translate object code to assembly language. Inverse of an assembler.

Folding: Procedure for randomizing the hash index. The upper and lower half of the key are added together before the index is calculated.

Horizontal table: A table whose entries are stored sequentially. That is, (entry one, byte one), (entry one, byte two), etc.

Hash index: The initial estimate of the location of an entry within the table.

Hashing: A nonlinear algorithm for storing/retrieving data from a table.

Hashing function: The algorithm or procedure for calculating the hash index.

Key: Field within an entry that is used to locate the entry. For example, surnames are the key field of the entries of a telephone directory.

Linear rehash: A method for resolving collisions. The table is searched sequentially from the point of impact.

Linear search: Table search which examines each item starting with the first item and proceeding sequentially.

MOD: Remainder of one number divided by another. That is, X MOD Y is the remainder of X/Y.

Pseudorandom rehash: A method for resolving collisions. A nonrepeating random number generator is used to determine the next entry to be searched.

Quadratic prime: A prime number which produces a remainder of 3 when divided by 4.

Quadratic rehash: A method for resolving collisions. A quadratic or second degree function is used to determine the next entry to be searched.

Rehash: Any algorithm for resolving collisions.

Squaring: Procedure for randomizing the hash index. The key is multiplied by itself before the hash index is computed.

Vertical table: A table where the bytes of each entry are stored sequentially. That is (entry one, byte one), (entry two, byte one), etc. FORTRAN stores arrays in this manner.

Weighted increment rehash: A method for resolving collisions. The hash index is used to determine the next entry to be searched.

BIBLIOGRAPHY

1. Day, Colin A, "Full Table Quadratic Searching for Scatter Storage," *Communications of the ACM* 13,8 (August 1970), pages 481-482.

2. Luccio, F, "Weighted Increment Linear Search for Scatter Tables," *Communications of the ACM* 15,12 (December 1972), pages 1045-1047.

3. Morris, Robert, "Scatter Storage Techniques," *Communications of the ACM* 11,1 (January 1968), pages 38-44.

4. Dollhoff, Terry, "Hashing Methods for Direct Access Files," Auerbach Computer Programming Management, Folio 15-02-02.

Improving Quadratic Rehash

John F Herbster

"Making Hash With Tables" by Terry Dollhoff *[BYTE, January 1977, page 18[1]]* is a good introduction to hash tables. However, quadratic methods for collision avoidance do not have to be complicated or suffer from "half table search." If the table length is a power of 2 and the quadratic increment is 3 as in the following simple and fast algorithm, then none of the table will be excluded from the search.

I have been using this scheme since about 1970 but have never seen it reported in the literature. Your readers may write to me for a copy of the proof that it works.■

A Quadratic Hash Table

The following algorithm assumes that the table length is a power of 2, the table words were initialized to VIRGIN, and MASK has a value equal to the table length minus 1.

1. Set DEL to 0.
2. Set I to hash code of KEY.
3. Let I=I.AND.MASK (ie: AND I with MASK).
4. If TABLE (I)=VIRGIN then go to NOTFOUND. (Note that TABLE(I) refers to the contents of location TABLE+I).
5. If TABLE(I)=KEY then go to FOUND.
6. Let DEL=(DEL+3).AND.MASK.
7. If DEL=0 then go to FULL. (Note that DEL gets back to 0 only after the whole table has been searched.)
8. Let I=(I+DEL).AND.MASK.
9. Go to step 4.

On return to the user's program via the NOTFOUND, or FOUND exits, the index, I, will point to the spot for a new table entry or the found entry respectively. The FULL return means the KEY was not found and that the table is full. Note that the value VIRGIN may not equal any possible value of KEY.

[1] *page 84 in this edition*

The Care and Feeding of Binary Trees

Cam Farnell

Many computer applications require sorting, searching, or both. While hashing is a useful tool for maintaining tables, it does not lend itself to particularly dense tables and is of no use in sorting. This article describes the use of binary trees to perform both sorting and searching at high speed with modest overhead.

Although numerous applications require sorting and searching, a good example that requires both is the label table in an assembler. During assembly, fast access is required each time a label is referenced, and, at the end, the table must be sorted if it is to be listed in alphabetical order.

Like all things, binary trees have advantages and disadvantages. They deal best with large amounts of data encountered in random order. This property makes them ideal for use in assemblers and other applications with similar data. Binary trees provide a method which is just the opposite of sequential searching or bubble sorting, which are at their best with small amounts of data.

Recognizing Binary Trees

The term 'binary tree' refers to a method of linking data records together in memory. Typically the records remain in memory in the same order in which they were encountered. Pointers are used to link them together in an ordered manner. The creation and use of these pointers is the heart of a binary tree.

In an assembler, for example, each label entry might contain the label itself, a one byte flag and a 16 bit value, as shown in figure 1. In order to be used as a binary tree, each entry must include two address pointers of 16 bits each, for a total of four bytes. These pointers are called *up* and *down* pointers since they will be used to point to higher and lower entries in the tree (more about that later). Our example now looks like figure 2.

In most discussions of trees, they are called 'trees' but are drawn upside down like roots. Others say this is silly and draw their trees right side up. In an effort to please all and offend none, trees are represented here on their sides. This has the advantage that *up* and *down* references, when applied to pointers, actually mean up and down in the figure, which is not true with the other representations.

A simple tree structure following our example would appear as in figure 3. For simplicity, only the label and the pointers have been shown. This tree is already built; we'll get to the logic on how to build one shortly.

Figure 1. Possible data arrangement for label entries for an assembler label table.

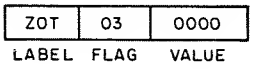


Figure 2. To use the binary tree structure, each entry must also have two pointers to point to the entry before it and the one after it.

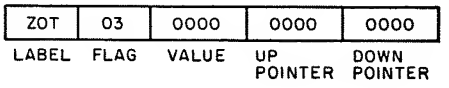
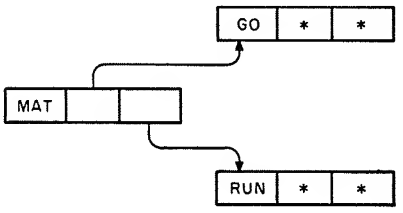


Figure 3. Simple binary tree structure. The first label encountered is MAT. This label leads up to GO or down to RUN, both of which end the tree.



The pointers link the tree together. The *up* pointer in each entry points to the rest of the tree which is above the current item while the *down* pointer points to those items below the current entry.

Null pointers are pointers which don't point anywhere. These are shown in the diagram as asterisks (*) and are usually represented in memory as zero. Null pointers indicate that a particular entry is the end of a branch.

Searching a Binary Tree

To search for an item in a binary tree, start with the first item (MAT in the example) and compare it to the item being looked for. If the item being searched is above the current entry in the sort sequence follow the *up* pointer, while if it is below, follow the *down* pointer. The procedure is repeated until either a match is found or a null pointer is encountered. The latter case indicates that the item searched for isn't in the tree. The logic for searching such a tree is shown in listing 1.

To search the tree for the word GO, start at MAT and perform a comparison. Since GO is above MAT, take the *up* pointer from MAT which points to GO, producing a match and stopping the search.

Growing Binary Trees

Once we have the routine to search a tree, adding items is easy. To add an item to the tree search it first to make sure the item isn't already there. Using the example above, first search the tree in order to add the label NUM. Comparing with MAT indicates that NUM is below, so follow the *down* pointer which points to RUN. Comparing with RUN indicates that NUM is above RUN, so follow the *up* pointer from RUN. But since the up pointer is a null pointer there is no match and the search is terminated. This null pointer, however, is the one that should point to our new entry. To add NUM simply build an entry for it at the end of the table (with both its pointers set to null) and then adjust the null *up* pointer from RUN to point to the new entry for NUM. The tree is now as shown in figure 4. The logic for adding a new item to the tree is shown in listing 2.

The procedure given above tells how to add items to an existing tree but doesn't tell how to get the tree started in the first place. To start a tree, take the first item to be added to the tree and build an entry for it with both its pointers set to null. Since it is the first item in the tree there are no pointers to it.

```
(search pointer = address of first tree entry)
DO UNTIL  (current entry = required entry)
   IF (current entry > required entry) THEN
      IF     (current up-pointer = 0) THEN
             (search pointer = address of current up-pointer)
             (return signaling 'not found')
      ELSE
             (search pointer = current up-pointer)
      ENDIF
   ELSE
      IF     (current down-pointer = 0) THEN
             (search pointer = address of current down-pointer)
             (return signaling 'not found')
      ELSE
             (search pointer = current down-pointer)
      ENDIF
   ENDIF
ENDDO
(return signaling 'found')
```

Listing 1. Binary tree search logic. This logical routine will search the entire binary tree until the looked for label is found or a null pointer is encountered. This listing expresses the logic in a "pseudo code" language.

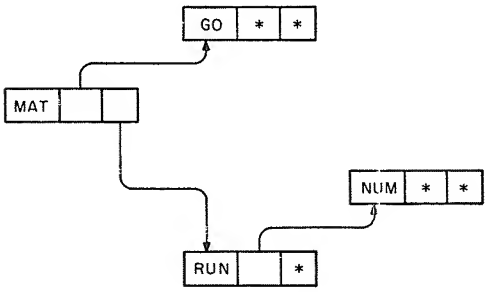


Figure 4. Inserting another entry to the binary tree. The up *pointer from RUN is now pointing to the new label which is lower than MAT but higher than RUN.*

Sorting with Binary Trees

So much for searching. What about sorting? Although it may not appear so, once a tree has been built it is actually sorted as well. This can be seen by placing a piece of paper over figure 4 and slowly sliding it down. The labels will appear in sequence because the diagram shows them in logical order. Reading them back from the computer's memory isn't quite this simple, since the logical order is given by the pointers rather than by physical order, but it's not very hard either.

The logic to read back a tree in order consists, in a nutshell, of performing the instructions in table 1.

The complete algorithm for this process is shown in listing 3. In order to remember the path followed a stack is used. Each stack entry requires 3 bytes: 2 for a 16 bit pointer and one for a flag to indicate if the path was via an *up* or *down* pointer.

```
(call the search routine)
IF (the item was found) THEN
   (return signaling 'duplicate')
ENDIF
(build an entry for the new item)
(get the pointer left by the search routine)
(use this pointer to store the address of the new entry...)
(... over the null pointer that terminated the search)
(return signaling completion)
```

Listing 2. Binary tree data insertion logic. This routine uses the search routine to find if the item is already in the tree. If it is not, the search routine returns with the pointer indicating the null pointer that ended the search. The new entry will be added to the tree at this position.

```
(search pointer = address of first tree entry)
DO FOREVER
   IF (current up-pointer = 0) THEN
      (print the current entry)
      DO WHILE (current down-pointer = 0)
         IF (stack is empty) THEN
            (return)
         ENDIF
         (search pointer = top pointer from stack)
         (search flag = top flag from stack)
         (pop the stack)
         DO WHILE (search flag = 'D')
            IF (stack is empty) THEN
               (return)
            ENDIF
            (search pointer = top pointer from stack)
            (search flag = top flag from stack)
            (pop the stack)
         ENDDO
         (print the current entry)
      ENDDO
      (push the stack)
      (top stack pointer = search pointer)
      (top stack flag = 'D')
      (search pointer = current down-pointer)
   ELSE
      (push the stack)
      (top stack pointer = search pointer)
      (top stack flag = 'U')
      (search pointer = current up-pointer)
   ENDIF
ENDDO
```

Listing 3. This list of instructions is the procedure that is followed when sorting a binary tree.

- Start at the root of the tree ('MAT' in the example).
- If there is an *up* pointer, follow it and make a record of the path followed.
- When returning from following the *up* pointer (or if the *up* pointer was null), print the current entry, then follow the *down* pointer and make a record of the path.
- When both pointers have been processed (or if the *down* pointer is null) back up to the previous entry.
- The sort is done when an attempt is made to back up to the previous entry and there is no previous entry found.

Table 1. Instructions for carrying out a sort of a binary tree.

- Start and the beginning of the tree (MAT).
- MAT has an *up* pointer, so follow it, pushing the address of MAT (the entry being come from) and a U flag (to indicate that the path followed an *up* pointer) onto the stack.
- The entry being pointed to now is GO.
- GO has a null *up* pointer which requires no action.
- Having processed GO's *up* pointer, GO is printed.
- GO also has a null *down* pointer which requires no action.
- Since both of GO's pointers have been processed, the stack is popped.
- The entry now being pointed to is MAT.
- Since a U flag was popped, the *up* pointer (but not the *down* pointer) has been examined; therefore MAT is printed.
- MAT has a valid *down* pointer, so it is followed and the address of MAT and a D flag (since the path was via a *down* pointer) are pushed onto the stack.
- The entry being pointed to is now RUN.
- RUN has a valid *up* pointer, so the address of RUN and a U flag are pushed onto the stack.
- The entry being pointed to is NUM.
- Since both of the pointers are null, NUM is printed and the stack is popped.
- The entry being pointed to is now RUN.
- Since RUN's *up* pointer has already been processed, RUN is printed.
- RUN's *down* pointer is null so the stack is again popped.
- MAT is now being pointed to.
- Having examined both the *up* and *down* pointers of MAT, an attempt is made to pop the stack.
- Since the stack is empty, it cannot be popped. This signals that all processing is complete.

Table 2. This is a trace of the procedure for reading back and printing the example tree of figure 4 using the logic routines of listing 3.

To illustrate this, table 2 is a trace of the procedure necessary to read back and print the example tree from figure 4.

This procedure will work for a tree of any size as long as there is enough room for the stack. The maximum number of stack entries required during the sorted readback depends on the order in which the data is placed in the tree. If the data comes in in random order, then the number of stack entries will not be much greater than the base 2 logarithm of the number of entries. For example, a stack with a depth of 16 should handle a tree containing up to about 64,000 entries. The worst case for stack depth occurs when the data was already sorted, or reverse sorted, when read in. This case will require as many stack entries as there are items in the tree.

Optimizing Binary Trees

If the tree routines use the lower 32 K bytes of address space, the sort readback stack can be reduced to two bytes per entry by using the high order address bit in place of the up or down flag. If this is done, the high order bit will probably have to be masked off before addresses from the stack are used to access memory.

It is possible to balance a binary tree as it is being built. Such a tree will always require a minimum number of comparisons for a search and a minimum amount of stack depth during sort readback. On the other hand, balancing a tree requires two more bits per entry (hence probably an extra byte) and is quite complex. The balancing algorithm is too complicated to include here; however, a complete description can be found in *The Art of Computer Programming*, Volume 3 by Donald Knuth. ■

About the Authors

David Higgins (Langston, Kitch and Associates Inc, 715 E 8th St, Topeka KS 66607) is a Senior Programmer, Analyst, and Instructor with Langston, Kitch and Associates, a consulting firm in the midwest. For the last three years he has been primarily involved with the development of software for a variety of minicomputers and in the development of computer packages to aid in the logical design of computer applications.

Dave has lectured nationwide on the subject of structured program design and is currently finishing a book on the application of structured program design methods to small computers.

Jerry Goff (Hewlett Packard, Colorado Springs Div., 1900 Garden of the Gods Rd, Colorado Springs CO 80907, (303) 598-1900) currently works for Hewlett Packard developing programs for automatic testing and analysis of analog hybrid microcircuits. While employed at Hewlett Packard, he obtained a Bachelor of Science degree in Physics from the University of New Mexico. He has also been a Production Engineer involved with the manufacture of thick film microelectronics.

His hobbies include general tinkering (both electrical and mechanical) and bicycling.

Albert D Hearn (98 SW 13th Ave, Boca Raton FL 33432) has worked for a large computer manufacturer as a programmer and programming manager for 12 years. He has helped develop and implement advanced programming methods for communications oriented processors.

Albert's hobbyist involvement includes designing and constructing a Signetics 2650 based system which he intends to use with a ham radio station.

Gregg Williams (1605 Eastmoreland #3, Memphis TN 38104) interests are varied as evidenced by his obtaining a combined BA in computer science and and the theatre. He also holds an MS degree in computer science from Memphis State University. Gregg's interests are divided equally between the arts and sciences since he feels this gives him a broader perspective on life.

Gregg is basically a software hobbyist. His related interests are artificial intelligence, simulation, graphics, and APL. He owns an OSI Challenger II with 20 K bytes of memory and graphics.

Tom Bohon (2215-A Walker Dr, Omaha NE 68123) is a captain in the Air Force assigned to the Airborne Branch, Force Control Programming Division, Directorate of Data Automation as an Airborne Data System Analyst. He is currently designing and implementing a data base information retrieval system for a minicomputer. Tom has a B in Mathematics from William Carey College. His programming languages experience includes COBOL, FORTRAN, BASIC, IBM SPS, JOVIAL and several assemblers.

Amateur photography fills Tom's spare time when he is not designing enhancements to his personal computer which is based on the National Semiconductor SC/MP-II microprocessor.

About the Authors, continued

Gary McGath (7 Silver Dr, Nashua NH 03060) is a professional computer programmer with a Bachelor of Science degree from MIT and a Master of Science from Purdue University. He is a native of New Hampshire and is currently employed at Itek.

Jim Butterfield (14 Brooklyn Av, Toronto Onatrio, Canada M4M 2X5) began programming real time computers back in the early 1960's. Since then, his work with Canadian National Telecommunications has led him away from programming as a career. Jim ardently welcomed the advent of microcomputers and became an enthusiastic hobbyist. In addition to developing a fast tape scheme for KIM known as Hypertape, Jim co-edited The First Book of KIM.

John Beetem (1809 Summit Av, Madison WI 53705) is a senior in Electrical and Computer Engineering at the University of Wisconsin in Madison. His programming portfolio includes: Q (a compiler), a graphics Star Trek type program with ships which look like the real thing, and a space shuttle simulation which is a three dimensional program wherein the user attempts landing on a field given the perspective view from inside the aircraft. All of the above are written in PDP-11 assembly language.

Terry Dollhoff (504 Early Fall Ct, Herndon VA 22070) is cofounder of the Rosse Corp, a microcomputer consulting firm. Although his recent work has centered around the TI 9900, he has designed hardware and software for many other microprocessors.

The hashing techniques discussed in his article were used by Terry while developing several assemblers and compilers.

John Herbster (Herbster Scientific, 3233 Mangum Rd, #100, Houston TX 77092) has obtained a Bachelor of Science degree in Physics from the Univer sity of Texas and studied graduate Electrical Engineering and Computer Science at Purdue. His work has taken him through environments ranging from the laboratory to a missile tracking ship.

In 1976 John started Herbster Scientific which has since been designing software and electronic interfaces, mainly for laboratory instrumentation.

Cam Farnell (RR # 1, Seeley's Bay, Ontario Canada) is employed as a programmer at MCM Computers in Kingston Ontario. Cam is presently constructing a house which has 40 conductor cable running through it for later computer connections. In his spare time he has written a word processor with which his article was prepared.

BYTE PUBLICATIONS, INC.
Editorial Director, Carl Helmers
Production Credits
Book Division Staff Participants:
Blaise Liffick, technical editor
Raymond G.A. Cote, assistant technical editor
Edmond Kelly Jr., production manager
Catherine Lahive, production assistant
Techart Associates, drafting
Walter Banks, University of Waterloo, CCNG, bar codes
George Banta Company, printing
Dawson Advertising Agency, cover art

Program Design

Vol. 1

The first of a new series of books, a collection of the best articles from BYTE Magazine and material never before published on the art and science of computer programming. The basic principle of this series is to provide the personal computer owner with some of the background information necessary to write and maintain programs effectively.

The series is designed to enhance the state of the personal computing art by encouraging readily understood, easily customized and fully designed programs.

This book introduces the subject of program design. The most critical part of developing a program is the design phase. Here most fatal errors are introduced and program specifications forgotten. It is also during this phase that errors are least costly to fix (both in terms of money and time), specifications are easiest to change, and program integrity simplest to insure. And this is true whether the program is the latest operating system or the newest computer game. This volume will help the personal computer user looking for a better way to design.

Other books in the Program Techniques series will include:

Simulation	ISBN 0-07-037826-6
Numbers in Theory and Practice	ISBN 0-07-037827-4
Bits and Pieces	ISBN 0-07-037828-2

ISBN 0-07-037825-8